

20241112-Week10 Nov月考、DFS模版/栈模拟 、NaraPan算法 & 线段树、树状数组

Updated 1435 GMT+8 Nov 12 2023

2024 fall, Complied by Hongfei Yan

Log:

2024/11/12 通常11月份前两周是各科密集期中考试时间，我们讲点拓展知识（线段树、树状数组），月考/作业相应降低难度，便于同学均衡各科学习时间。

2024/11/09 部分内容取自, https://github.com/GMyhf/2023fall-cs101/blob/main/20231031_SegmentTree_BIT.md

Recap

R1. 题目

E07618: 病人排队

sorttngs, <http://cs101.openjudge.cn/practice/07618/>

病人登记看病，编写一个程序，将登记的病人按照以下原则排出看病的先后顺序：

1. 老年人（年龄 ≥ 60 岁）比非老年人优先看病。
2. 老年人按年龄从大到小的顺序看病，年龄相同的按登记的先后顺序排序。
3. 非老年人按登记的先后顺序看病。

输入

第1行，输入一个小于100的正整数，表示病人的个数；

后面按照病人登记的先后顺序，每行输入一个病人的信息，包括：一个长度小于10的字符串表示病人的ID（每个病人的ID各不相同且只含数字和字母），一个整数表示病人的年龄，中间用单个空格隔开。

输出

按排好的看病顺序输出病人的ID，每行一个。

把老人和年轻人用两个列表储存，如果不确定sort是不是稳定的可以自行编入一个数据然后用sort输出看看顺序会不会变。有一个坑点是b要提前转成int，否则后续排序的时候会按照b的字典序而不是大小来排序。

```
1 # Read the number of patients
2 n = int(input())
3
4 # Initialize lists for elderly and non-elderly patients
5 elderly = []
6 non_elderly = []
7
8 # Read patient information
9 for _ in range(n):
10     patient_id, age = input().split()
11     age = int(age)
12     if age >= 60:
13         elderly.append((patient_id, age))
14     else:
15         non_elderly.append((patient_id, age))
16
17 # Sort elderly patients by age in descending order
18 elderly.sort(key=lambda x: -x[1])
19
20 # Concatenate elderly and non-elderly lists
21 sorted_patients = elderly + non_elderly
```

```

22
23 # Print the sorted patient IDs
24 for patient in sorted_patients:
25     print(patient[0])

```

Python 自带的 list.sort() 方法和内置的 sorted() 函数都是使用 Timsort 算法实现的。Timsort 是一种混合排序算法，源自归并排序和插入排序，由 Tim Peters 在 2002 年为 Python 编程语言发明。它是一种稳定的排序算法。

稳定性意味着如果两个元素具有相同的值，它们在排序后的列表中的相对位置不会改变。例如，如果有一个列表，其中包含多个具有相同值的元素，稳定排序会确保这些元素在排序后保持原有的顺序。

Merge sort很重要，有的题目就是在考merge sort。例如：07622:求排列的逆序数，<http://cs101.openjudge.cn/practice/07622/>

E23555: 节省存储的矩阵乘法

implementation, matrices, <http://cs101.openjudge.cn/practice/23555/>

由于矩阵存储非常耗费空间，一个长度n宽度m的矩阵需要花费n*m的存储，因此我们选择用另一种节省空间的方法表示矩阵。一个矩阵X可以表示为三元组的序列，每个三元组代表（行号，列号，元素值），如果元素值是0则我们不存储这个三元组，这样对于0很多的大型矩阵，我们节省了很多存储空间。现在我们有两个用这种方式表示的矩阵X和Y，我们想要计算这两个矩阵的乘积，并且也用三元组形式表达，该如何完成呢。

如果不知道矩阵如何相乘，可以参考：<http://cs101.openjudge.cn/practice/18161>

输入

输入第一行是三个整数n, m1, m2, 两个矩阵X, Y的维度都是n*n, m1是矩阵X中的非0元素数, m2是矩阵Y中的非0元素数。

之后是m1行，每行是一个三元组（行号，列号，元素值），代表X矩阵的元素值，注意行列编号都从0开始。

之后是m2行，每行是一个三元组（行号，列号，元素值），代表Y矩阵的元素值，注意行列编号都从0开始。

输出

输出是m3行，代表X和Y两个矩阵乘积中的非0元素的数目，按照先行号后列号的方式递增排序。

每行仍然是前述的三元组形式。

18161:矩阵运算。<http://cs101.openjudge.cn/practice/18161>

矩阵乘法运算必须要前一个矩阵的列数与后一个矩阵的行数相同，

如m行n列的矩阵A与n行p列的矩阵B相乘，可以得到m行p列的矩阵C，

矩阵C的每个元素都由A的对应行中的元素与B的对应列中的元素一一相乘并求和得到，

即 $c_{ij} = \sum a_{ik} b_{kj}$

(c_{ij} 表示C矩阵中第i行第j列元素)。

即, $c_{ij} = \sum a_{ik} b_{kj}$

输入放到矩阵里面就好了，在计算乘法之后一旦有值不是0就可以在遍历中直接把位置和值输出。

```

1 # 汤伟杰, 24信息管理系
2 n, m1, m2 = map(int, input().split())
3 a = [[0] * n for _ in range(n)]
4 b = [[0] * n for _ in range(n)]
5 for _ in range(m1):
6     x, y, v = map(int, input().split())
7     a[x][y] = v
8 for _ in range(m2):
9     x, y, v = map(int, input().split())
10    b[x][y] = v
11 c = [[0] * n for _ in range(n)]
12 for i in range(n):
13     for j in range(n):
14         c[i][j] = sum(a[i][k] * b[k][j] for k in range(n))
15         if c[i][j] != 0:
16             print(i, j, c[i][j])

```

使用字典来存储稀疏矩阵。读取 `m2` 行输入，每行包含三个整数 `i, j, val`，表示矩阵 `y` 中第 `i` 行第 `j` 列的元素值为 `val`。注意这里将 `y` 存储为转置形式，即 `y[j][i]` 而不是 `y[i][j]`，这是为了方便后续的矩阵乘法计算。

```

1 # 焦玮宸 24数学科学学院
2 n, m1, m2 = map(int, input().split())
3 X, Y = [{ } for i in range(n)], [{ } for i in range(n)]
4 for _ in range(m1):
5     i, j, val = map(int, input().split())
6     X[i][j] = val
7 for _ in range(m2):
8     i, j, val = map(int, input().split())
9     Y[j][i] = val
10 for i in range(n):
11     for j in range(n):
12         res = 0
13         for ind in range(n):
14             if ind in X[i] and ind in Y[j]:
15                 res += X[i][ind] * Y[j][ind]
16         if res:
17             print(i, j, res)

```

M18182: 打怪兽

implementation/sortings/data structures, <http://cs101.openjudge.cn/practice/18182/>

Q神无聊的时候经常打怪兽。现在有一只怪兽血量是 b , Q神在一些时刻可以选择一些技能打怪兽, 每次释放技能都会让怪兽掉血。

现在给出一些技能 t_i, x_i , 代表这个技能可以在 t_i 时刻使用, 并且使得怪兽的血量下降 x_i 。这个打怪兽游戏有个限制, 每一时刻最多可以使用 m 个技能 (一个技能只能用一次)。如果技能使用得当, 那么怪兽会在哪一时刻死掉呢?

输入

第一行是数据组数 $nCases$, $nCases \leq 100$

对于每组数据, 第一行是三个整数 n, m, b , n 代表技能的个数, m 代表每一时刻可以使用最多 m 个技能, b 代表怪兽初始的血量。

$1 \leq n \leq 1000$, $1 \leq m \leq 1000$, $1 \leq b \leq 10^9$

接下来 n 行, 每一行一个技能 t_i, x_i , $1 \leq t_i \leq 10^9$, $1 \leq x_i \leq 10^9$

输出

对于每组数据, 输出怪兽在哪一时刻死掉, 血量小于等于0就算挂, 如果不能杀死怪兽, 输出alive

用了字典, 注意字典本身无序, 只能对 keys 或 values 排序。

用defaultdict, 就不用判断key是否在字典中了。defaultdict会自动为这个键创建一个默认值。

```
1 from collections import defaultdict
2
3 cases = int(input())
4
5 for _ in range(cases):
6     situation = "alive"
7     n, m, b = map(int, input().split())
8     a = defaultdict(list)
9
10    # Input coordinates
11    for _ in range(n):
12        x, y = map(int, input().split())
13        a[x].append(y)
14
15    # Process coordinates
16    for x in sorted(a):
17        if m >= len(a[x]):
18            b -= sum(a[x])
19        else:
20            a[x].sort(reverse=True)
21            b -= sum(a[x][:m])
22        if b <= 0:
23            situation = x
24            break
25
26 print(situation)
```

如果 m 比较大的话，`heapq.nlargest` 可能会比较慢。可以考虑在收集数据时直接维护一个大小为 m 的堆。

```
1 # 韩博文, 24城市与环境学院
2 from collections import defaultdict
3 import heapq
4
5 for _ in range(int(input())):
6     n, m, b = map(int, input().split())
7     d = defaultdict(list)
8
9     # Collect all the damage values for each time point
10    for _ in range(n):
11        t, x = map(int, input().split())
12        d[t].append(x)
13
14    # Calculate the total damage for each time point using a heap
15    for t in d:
16        if len(d[t]) > m:
17            d[t] = sum(heapq.nlargest(m, d[t]))
18        else:
19            d[t] = sum(d[t])
20
21    # Sort the time points by their occurrence
22    dp = sorted(d.items())
23
24    # Apply the damage and check if the blood is depleted
25    for t, damage in dp:
26        b -= damage
27        if b <= 0:
28            print(t)
29            break
30    else:
31        print('alive')
```

在收集数据时直接维护一个大小为 m 的堆，这样可以减少后续的计算开销。如果堆的大小已经达到 m ，则使用 `heapq.heappushpop` 将新值加入堆，并弹出最小值。这样可以确保堆中始终保留最大的 m 个值。

```
1 # 韩博文, 24城市与环境学院
2 from collections import defaultdict
3 import heapq
4
5 for _ in range(int(input())):
6     n, m, b = map(int, input().split())
7     d = defaultdict(list)
8
9     # Collect all the damage values for each time point and maintain a heap of size m
10    for _ in range(n):
11        t, x = map(int, input().split())
12        if len(d[t]) < m:
```

```

13     heapq.heappush(d[t], x)
14 else:
15     heapq.heappushpop(d[t], x)
16
17 # Calculate the total damage for each time point
18 for t in d:
19     d[t] = sum(d[t])
20
21 # Sort the time points by their occurrence
22 dp = sorted(d.items())
23
24 # Apply the damage and check if the blood is depleted
25 for t, damage in dp:
26     b -= damage
27     if b <= 0:
28         print(t)
29         break
30 else:
31     print('alive')

```

M28780: 零钱兑换3

dp, <http://cs101.openjudge.cn/practice/28780/>

给定一组n种不同面额的硬币，以及要支付的总金额

计算并返回可以凑成总金额所需的 **最少的硬币个数**。如果没有任何一种硬币组合能组成总金额，返回 -1。

你可以认为每种硬币的数量是无限的。

输入

输入为两行

第一行为两个整数n ($1 \leq n \leq 100$) , m ($0 \leq m \leq 10^6$) , 其中n表示硬币的种类数, m表示要凑的总金额

第二行为n个整数, 表示硬币的面值

输出

可以凑成总金额所需的最少的硬币个数。如果没有任何一种硬币组合能组成总金额，则输出 -1。

样例输入

```

1 sample1 input:
2 3 11
3 1 2 4
4
5 sample1 output:
6 4

```

样例输出

```
1 sample2 input:  
2 1 3  
3 2  
4  
5 sample2 output:  
6 -1
```

提示

dp

来源

2024 TA-Lhw

完全背包，类似于剪彩 189A. Cut Ribbon, <https://codeforces.com/problemset/problem/189/A>

时间：16875ms

```
1 n, m = map(int, input().split())  
2 coins = list(map(int, input().split()))  
3 dp = [float("inf")] * (m + 1)  
4 dp[0] = 1  
5 for i in coins:  
6     dp[i] = 1  
7 for i in range(1, m + 1):  
8     for coin in coins:  
9         if i - coin >= 0:  
10             dp[i] = min(dp[i], dp[i - coin] + 1)  
11  
12 #print(dp)  
13 if dp[m] == float("inf"):  
14     print(-1)  
15 else:  
16     print(dp[m])
```

```
1 from math import inf  
2 n, m = map(int, input().split())  
3 coins = list(map(int, input().split()))  
4 dp = [0] + [inf for _ in range(m)]  
5 for i in range(n):  
6     for j in range(coins[i], m + 1):  
7         dp[j] = min(dp[j], dp[j - coins[i]] + 1)  
8 print(dp[m] if dp[m] != inf else -1)
```

时间: 5210ms

```
1 # 2400010989 韩宇宸 工学院
2 import bisect
3
4 # 读取输入
5 n, m = map(int, input().split())
6 face = sorted(map(int, input().split())) # 直接排序后使用
7 coins = [float('inf')] * (m + 1)
8 coins[0] = 0 # 初始值
9
10 # 动态规划计算最小硬币数
11 for i in range(1, m + 1):
12     w = bisect.bisect_right(face, i)
13
14     #for k in range(w):
15     #    coins[i] = min(coins[i], coins[i - face[k]] + 1)
16     if w != 0:
17         coins[i] = min(coins[i - face[k]] for k in range(w)) + 1
18
19 # 输出结果
20 print(coins[m] if coins[m] != float('inf') else -1)
```

T12757: 阿尔法星人翻译官

implementation, <http://cs101.openjudge.cn/practice/12757>

阿尔法星人为了了解地球人，需要将地球上所有的语言转换为他们自己的语言，其中一个小模块是要将地球上英文表达的数字转换为阿尔法星人也理解的阿拉伯数字。请你为外星人设计这个模块，即给定一个用英文表示的整数，将其转换成用阿拉伯数字表示的整数。这些数的范围从-999,999,999到+999,999,999。下列单词是你的程序中将遇到的所有有关数目的英文单词：

negative, zero, one, two, three, four, five, six, seven, eight, nine, ten, eleven, twelve, thirteen, fourteen, fifteen, sixteen, seventeen, eighteen, nineteen, twenty, thirty, forty, fifty, sixty, seventy, eighty, ninety, hundred, thousand, million

输入

输入一行，由几个表示数目的英文单词组成(长度不超多200)。注意：负号将由单词negative表示。

当数的大小超过千时，并不用完全单词hundred表示。例如1600将被写为"one thousand six hundred"，而不是"sixteen hundred"。

输出

输出一行，表示答案。

这题就恶心在 hundred 可能在千和百万之前，所以要暂时储存出现的一百，等后边出现千或百万的时候就好处理了。

```
1 tokens = [str(i) for i in input().split()]
2 dic={"zero":0, "one":1, "two":2, "three":3, "four":4, "five":5, "six":6,
3     "seven":7, "eight":8, "nine":9, "ten":10, "eleven":11, "twelve":12,
4     "thirteen":13, "fourteen":14, "fifteen":15, "sixteen":16, "seventeen":17,
5     "eighteen":18, "nineteen":19, "twenty":20, "thirty":30, "forty":40,
6     "fifty":50, "sixty":60, "seventy":70, "eighty":80, "ninety":90,
7     "hundred":100, "thousand":1000, "million":1000000}
8
9 sign = 1
10 if tokens[0]=="negative":
11     sign = -1
12     del tokens[0]
13
14 total = 0
15 tmp = 0
16 for i in tokens:
17     if i in ("thousand", "million"):
18         total += tmp*dic[i]
19         tmp = 0
20         continue
21     if i == "hundred":
22         tmp *= dic[i]
23     else:
24         tmp += dic[i]
25
26 print( sign * (total + tmp) )
```

2021fall-cs101，李佳霖。对上面程序解读：可以把hundred, thousand, million 这些看成是计量单位，而其他的具体的数当作系数。由于这道题不需要考虑如one thousand million 而只存在one hundred million 这样的情况，因此可以把thousand 和million 看成是一类。tmp 作为累计计数单位，对具体的数字进行加和处理。而遇到 hundred 时便进行释放，成100 处理；遇到thousand 和million则需要考虑前面是否存在hundred million 这样的情况，并做对应的加和处理。最终输出带有正负号的数字。

2021fall-cs101，龚靖淞。one thousand million就是one billion来着。

2021fall-cs101，侯勇启。思路：用字典实现即可，易知不会出现thousand million 这样的数据，只会存在 hundred million 和hundred thousand 数据，从而每次遍历到thousand、million 都可以结算；用cnt 滚动记录阶段值，结算后清零。

递归实现

```
1 # 焦玮宸 24数学科学学院
```

```

2 dictionary = {'zero': 0, 'one': 1, 'two': 2, 'three': 3, 'four': 4, 'five': 5, 'six': 6, 'seven': 7, 'eight': 8, 'nine': 9, 'ten': 10, 'eleven': 11, 'twelve': 12, 'thirteen': 13, 'fourteen': 14, 'fifteen': 15, 'sixteen': 16, 'seventeen': 17, 'eighteen': 18, 'nineteen': 19, 'twenty': 20, 'thirty': 30, 'forty': 40, 'fifty': 50, 'sixty': 60, 'seventy': 70, 'eighty': 80, 'ninety': 90}
3 def convert(words):
4     if words[0] == "negative":
5         return -convert(words[1:])
6     if "million" in words:
7         ind = words.index("million")
8         return convert(words[:ind]) * (10 ** 6) + (convert(words[ind + 1:])) if ind <
len(words) - 1 else 0
9     if "thousand" in words:
10        ind = words.index("thousand")
11        return convert(words[:ind]) * (10 ** 3) + (convert(words[ind + 1:])) if ind <
len(words) - 1 else 0
12    if "hundred" in words:
13        ind = words.index("hundred")
14        return convert(words[:ind]) * (10 ** 2) + (convert(words[ind + 1:])) if ind <
len(words) - 1 else 0
15    return sum(list(map(lambda s: dictionary[s], words)))
16
17
18 print(convert(list(input().split())))

```

T16528: 充实的寒假生活

greedy/dp, cs10117 Final Exam, <http://cs101.openjudge.cn/practice/16528/>

寒假马上就要到了，龙傲天同学获得了从第0天开始到第60天结束为期61天超长寒假，他想要尽可能丰富自己的寒假生活。

现提供若干个活动的起止时间，请计算龙同学这个寒假至多可以参加多少个活动？注意所参加的活动不能有任何时间上的重叠，在第x天结束的活动和在第x天开始的活动不可同时选择。

输入

第一行为整数n，代表接下来输入的活动个数($n < 10000$)

紧接着的n行，每一行都有两个整数，第一个整数代表活动的开始时间，第二个整数代表全结束时间

输出

输出至多参加的活动个数

区间问题，Greedy。按照结束时间排序，然后看找下一个起始时间不超过结束时间的区间，一直贪心。

```

1 n = int(input())
2 events = [list(map(int, input().split())) for _ in range(n)]
3 cur, ans = -1, 0
4 for event in sorted(events, key=lambda x: x[1]):
5     if event[0] > cur:
6         cur = event[1]; ans += 1
7 print(ans)

```

数据量过小，可以 $O(nm)$ dp。这个+1，设置的很巧妙。

```

1 # 高景行 24数学科学学院
2 n = int(input())
3 a = []
4 for i in range(n):
5     x, y = map(int, input().split())
6     a.append((x + 1, y + 1))
7 a = sorted(a, key = lambda _:_[0])
8 dp = [0] * 65
9 for i in range(n):
10    for j in range(a[i][1], 62):
11        dp[j] = max(dp[j], dp[a[i][0] - 1] + 1)
12 print(dp[61])

```

R2. 学习总结和收获

焦玮宸 数学科学学院。Assignment #7: Nov Mock Exam立冬

复习了一下 dp 的一些基础算法。

额外练习：每日选做的所有题，以及 LeetCode 上一些题目，比如 509, 70, 62, 1137, 650, 264, 279, 343, 416, 494, 1049。

1 DFS模版+用栈来模拟递归

示例02386: Lake Counting

dfs similar, <http://cs101.openjudge.cn/practice/02386>

Due to recent rains, water has pooled in various places in Farmer John's field, which is represented by a rectangle of $N \times M$ ($1 \leq N \leq 100$; $1 \leq M \leq 100$) squares. Each square contains either water ('W') or dry land ('.'). Farmer John would like to figure out how many ponds have formed in his field. A pond is a connected set of squares with water in them, where a square is considered adjacent to all eight of its neighbors.

Given a diagram of Farmer John's field, determine how many ponds he has.

输入

* Line 1: Two space-separated integers: N and M

* Lines 2.. $N+1$: M characters per line representing one row of Farmer John's field. Each character is either 'W' or '.'. The characters do not have spaces between them.

输出

* Line 1: The number of ponds in Farmer John's field.

样例输入

```
1 10 12
2 W.....WW.
3 .WWW.....WWW
4 ....WW...WW.
5 .....WW.
6 .....W..
7 ..W.....W..
8 .W.W.....WW.
9 W.W.W.....W.
10 .W.W.....W.
11 ..W.....W.
```

样例输出

```
1 | 3
```

提示

OUTPUT DETAILS:

There are three ponds: one in the upper left, one in the lower left, and one along the right side.

来源: USACO 2004 November

```

1 #1.dfs
2 import sys
3
4 # input = sys.stdin.read
5 sys.setrecursionlimit(20000)
6
7
8 def dfs(x, y):
9     # 标记当前位置为已访问
10    field[x][y] = '.'
11
12    for dx, dy in directions:
13        nx, ny = x + dx, y + dy
14
15        # 检查新位置是否在地图范围内且未被访问
16        if 0 <= nx < n and 0 <= ny < m and field[nx][ny] == 'W':
17            dfs(nx, ny)
18
19
20    # 一次性读取所有输入
21    # data = input().split()
22    # n, m = map(int, data[:2])
23    # field = [list(row) for row in data[2:2+n]]
24    n, m = map(int, input().split())
25    field = [list(input()) for _ in range(n)]
26
27    # 初始化8个方向
28    directions = [(-1, -1), (-1, 0), (-1, 1), (0, -1), (0, 1), (1, -1), (1, 0), (1, 1)]
29
30    # 遍历地图
31    for i in range(n):
32        for j in range(m):
33            if field[i][j] == 'W':
34                dfs(i, j)
35                cnt += 1
36
37    print(cnt)
38

```

通过使用栈来模拟递归，可以避免因递归过深导致的栈溢出问题。

```

1 def dfs(x, y):
2     stack = [(x, y)]
3     while stack:
4         x, y = stack.pop()
5         if field[x][y] != 'W':
6             continue

```

```

7     field[x][y] = '.' # 标记当前位置为已访问
8     for dx, dy in directions:
9         nx, ny = x + dx, y + dy
10        if 0 <= nx < n and 0 <= ny < m and field[nx][ny] == 'W':
11            stack.append((nx, ny))
12
13 # 读取输入
14 n, m = map(int, input().split())
15 field = [list(input()) for _ in range(n)]
16
17 # 初始化8个方向
18 directions = [(-1, -1), (-1, 0), (-1, 1), (0, -1), (0, 1), (1, -1), (1, 0), (1, 1)]
19
20 # 计数器
21 cnt = 0
22
23 # 遍历地图
24 for i in range(n):
25     for j in range(m):
26         if field[i][j] == 'W':
27             dfs(i, j)
28             cnt += 1
29
30 print(cnt)

```

示例05585: 晶矿的个数

matrices/dfs similar, <http://cs101.openjudge.cn/practice/05585>

在某个区域发现了一些晶矿，已经探明这些晶矿总共有分为两类，为红晶矿和黑晶矿。现在要统计该区域内红晶矿和黑晶矿的个数。假设可以用二维地图m[][]来描述该区域，若m[i][j]为#表示该地点是非晶矿地点，若m[i][j]为r表示该地点是红晶矿地点，若m[i][j]为b表示该地点是黑晶矿地点。一个晶矿是由相同类型的并且上下左右相通的晶矿点组成。现在给你该区域的地图，求红晶矿和黑晶矿的个数。

输入

第一行为k，表示有k组测试输入。

每组第一行为n，表示该区域由n*n个地点组成， $3 \leq n \leq 30$

接下来n行，每行n个字符，表示该地点的类型。

输出

对每组测试数据输出一行，每行两个数字分别是红晶矿和黑晶矿的个数，一个空格隔开。

样例输入

```
1 2
2 6
3 r##bb#
4 ###b##
5 #r##b#
6 #r##b#
7 #r#####
8 ######
9 4
10 #####
11 #rrb
12 #rr#
13 ##bb
```

样例输出

```
1 2 2
2 1 2
```

```
1 dire = [[-1,0], [1,0], [0,-1], [0,1]]
2
3 def dfs(x, y, c):
4     m[x][y] = '#'
5     for i in range(len(dire)):
6         tx = x + dire[i][0]
7         ty = y + dire[i][1]
8         if m[tx][ty] == c:
9             dfs(tx, ty, c)
10
11 for _ in range(int(input())):
12     n = int(input())
13     m = [[0 for _ in range(n+2)] for _ in range(n+2)]
14
15     for i in range(1, n+1):
16         m[i][1:-1] = input()
17
18     r = 0 ; b=0
19     for i in range(1, n+1):
20         for j in range(1, n+1):
21             if m[i][j] == 'r':
22                 dfs(i, j, 'r')
23                 r += 1
24             if m[i][j] == 'b':
25                 dfs(i,j,'b')
26                 b += 1
27
28 print(r, b)
```

通过使用栈来模拟递归，可以避免因递归过深导致的栈溢出问题。

```
1 import sys
2
3 # 定义方向数组
4 dire = [[-1, 0], [1, 0], [0, -1], [0, 1]]
5
6 def dfs(x, y, c):
7     stack = [(x, y)]
8     while stack:
9         x, y = stack.pop()
10        if m[x][y] != c:
11            continue
12        m[x][y] = '#'
13        for dx, dy in dire:
14            tx, ty = x + dx, y + dy
15            if m[tx][ty] == c:
16                stack.append((tx, ty))
17
18 # 处理多组测试数据
19 t = int(input())
20 for _ in range(t):
21     n = int(input())
22
23     # 初始化矩阵并添加边界
24     m = [['.'] * (n + 2) for _ in range(n + 2)]
25     for i in range(1, n + 1):
26         m[i][1:-1] = list(input().strip())
27
28     r = 0
29     b = 0
30
31     # 遍历矩阵
32     for i in range(1, n + 1):
33         for j in range(1, n + 1):
34             if m[i][j] == 'r':
35                 dfs(i, j, 'r')
36                 r += 1
37             elif m[i][j] == 'b':
38                 dfs(i, j, 'b')
39                 b += 1
40
41     print(r, b)
```

示例23937: 逃出迷宫

<http://cs101.openjudge.cn/practice/23937/>

"Boom!" 小锅一觉醒来发现自己落入了一个 $N \times N$ ($2 \leq N \leq 20$)的迷宫之中，为了逃出这座迷宫，小锅需要从左上角(0, 0)处的入口跑到右下角($N-1, N-1$)处的出口逃出迷宫。由于小锅每一步都想缩短和出口之间的距离，所以他只会向右和向下走。假设我们知道迷宫的地图（以0代表通路，以1代表障碍），请你编写一个程序，判断小锅能否从入口跑到出口？

例如，对于下图所示的迷宫：

0	0	1	1	0
0	0	0	0	0
0	1	1	1	0
0	1	1	1	0
0	1	1	1	0

小锅可以如下图红线所示从迷宫左上角的入口抵达迷宫右下角的出口：

0	0	1	1	0
0	0	0	0	0
0	1	1	1	0
0	1	1	1	0
0	1	1	1	0

输入

第一行为一个整数 N ，代表迷宫的大小

接下来 N 行为迷宫地图，迷宫地块之间以空格分隔

输入保证(0, 0)和($N - 1, N - 1$)处可以通过

输出

一行字符串，如果能跑到出口则输出Yes，否则输出No

样例输入

```
1 5
2 0 0 1 1 0
3 0 0 0 0 0
4 0 1 1 1 0
5 0 1 1 1 0
6 0 1 1 1 0
```

样例输出

```
1 Yes
```

提示

用递归解。设计函数ok(r,c)，返回True或False，表示从位置(r,c)出发能否走到终点。
从(r,c) 出发可以想办法往前走一步，然后看问题变成什么

题目说了只能走到0的格子，不能走到1的格子

这是模版题目，涉及到 递归/dfs/回溯。一旦出现模版题目，最多是中等难度，要求必须会。

```
1 def dfs(mx, visited, x, y):
2     # 如果到达右下角，返回True
3     if x == n - 1 and y == n - 1:
4         return True
5
6     # 定义向右和向下的移动方向
7     directions = [(0, 1), (1, 0)]
8
9     for dx, dy in directions:
10        nx = x + dx
11        ny = y + dy
12        # 检查新坐标是否在矩阵范围内，是否已经访问过，以及是否可以通过
13        if 0 <= nx < n and 0 <= ny < n and not visited[nx][ny] and mx[nx][ny] == 0:
14            visited[nx][ny] = True
15            if dfs(mx, visited, nx, ny):
16                return True
17            visited[nx][ny] = False
18
19    return False
20
21 # 读取输入
22 n = int(input())
23 mx = [list(map(int, input().split())) for _ in range(n)]
24
25 # 初始化访问标记数组
26 visited = [[False] * n for _ in range(n)]
27
28 # 起始点 (0, 0) 必须是可以通过的
29 if mx[0][0] == 1:
```

```

30     print('No')
31 else:
32     visited[0][0] = True
33     if dfs(mx, visited, 0, 0):
34         print('Yes')
35     else:
36         print('No')

```

通过使用栈来模拟递归，可以避免因递归过深导致的栈溢出问题。

```

1 # 读取输入
2 n = int(input())
3 mx = [list(map(int, input().split())) for _ in range(n)]
4
5 # 初始化访问标记数组
6 visited = [[False] * n for _ in range(n)]
7
8 # 起始点 (0, 0) 必须是可以通过的
9 if mx[0][0] == 1:
10     print('No')
11 else:
12     # 使用栈来模拟递归
13     stack = [(0, 0)]
14     visited[0][0] = True
15
16     while stack:
17         x, y = stack.pop()
18
19         # 如果到达右下角，返回True
20         if x == n - 1 and y == n - 1:
21             print('Yes')
22             break
23
24         # 定义向右和向下的移动方向
25         directions = [(0, 1), (1, 0)]
26
27         for dx, dy in directions:
28             nx = x + dx
29             ny = y + dy
30             # 检查新坐标是否在矩阵范围内，是否已经访问过，以及是否可以通过
31             if 0 <= nx < n and 0 <= ny < n and not visited[nx][ny] and mx[nx][ny] ==
32             0:
33                 visited[nx][ny] = True
34                 stack.append((nx, ny))
35             else:
36                 print('No')

```

2 Narayana Pandita's algorithm & Cantor Expansion

示例 01833: 排列

Math, <http://cs101.openjudge.cn/practice/01833>

大家知道，给出正整数n，则1到n这n个数可以构成 $n!$ 种排列，把这些排列按照从小到大的顺序（字典顺序）列出，如 n=3 时，列出1 2 3, 1 3 2, 2 1 3, 2 3 1, 3 1 2, 3 2 1六个排列。

任务描述：

给出某个排列，求出这个排列的下k个排列，如果遇到最后一个排列，则下1排列为第1个排列，即排列1 2 3...n。比如：n = 3, k=2 给出排列2 3 1，则它的下1个排列为3 1 2，下2个排列为3 2 1，因此答案为3 2 1。

输入

第一行是一个正整数m，表示测试数据的个数，下面是m组测试数据，每组测试数据第一行是2个正整数n(1 <= n < 1024)和k(1 <= k <= 64)，第二行有n个正整数，是1, 2 ... n的一个排列。

输出

对于每组输入数据，输出一行，n个数，中间用空格隔开，表示输入排列的下k个排列。

样例输入

1	3
2	3 1
3	2 3 1
4	3 1
5	3 2 1
6	10 2
7	1 2 3 4 5 6 7 8 9 10

样例输出

1	3 1 2
2	1 2 3
3	1 2 3 4 5 6 7 9 8 10

来源

qinlu@POJ

这三个题目是相同的，tags: two pointers

01833: 排列

<http://cs101.openjudge.cn/practice/01833/>

02996: 选课

<http://cs101.openjudge.cn/practice/02996/>

LeetCode31.下一个排列

<https://leetcode.cn/problems/next-permutation/>

1.1 思路一： Narayana Pandita's algorithm (Next Permutation)

Wikipedia has a nice article on lexicographical order generation. It also describes an algorithm to generate the next permutation.

https://en.wikipedia.org/wiki/Permutation#Generation_in_lexicographic_order

Quoting:

The method goes back to Narayana Pandita in 14th century India, and has been rediscovered frequently.

[https://en.wikipedia.org/wiki/Narayana_Pandita_\(mathematician\)](https://en.wikipedia.org/wiki/Narayana_Pandita_(mathematician))

Narayana's cows sequence

$$N_n = N_{n-1} + N_{n-3} \text{ for } n > 2,$$

with initial values

$$N_0 = N_1 = N_2 = 1.$$

The first few terms are 1, 1, 1, 2, 3, 4, 6, 9, 13, 19, 28, 41, 60, 88,... (sequence [A000930](#) in the OEIS).

The On-Line Encyclopedia of Integer Sequences (OEIS), <https://oeis.org/>

The following algorithm generates the next permutation lexicographically after a given permutation. It changes the given permutation in-place.

1. Find the highest index i such that $s[i] < s[i+1]$. If no such index exists, the permutation is the last permutation.
2. Find the highest index $j > i$ such that $s[j] > s[i]$. Such a j must exist, since $i+1$ is such an index.
3. Swap $s[i]$ with $s[j]$.
4. Reverse the order of all of the elements after index i till the last element.

以下算法生成给定排列的下一个字典序排列。该算法会就地修改给定的排列。

1. 找到最高的索引 i , 使得 $s[i] < s[i+1]$ 。如果不存在这样的索引, 则该排列已经是最后一个排列。
2. 找到最高的索引 $j > i$, 使得 $s[j] > s[i]$ 。这样的 j 必然存在, 因为 $i+1$ 就是一个这样的索引。
3. 交换 $s[i]$ 和 $s[j]$ 。
4. 反转索引 i 之后的所有元素, 直到最后一个元素。

即：

- 1) 从后往前找第一组相邻的升序数对，记录左边的位置p。
- 2) 从后往前找第一个比p位置的数大的数，将两个数交换。
- 3) 把p位置后所有数字逆序。

举例：

1. 从数列的右边向左寻找连续递增序列，例如对于：1,3,5,4,2，其中5-4-2即为递增序列。
2. 从上述序列中找一个比它前面的数（3）大的最小数（4），并将且交换这两个数。于是1,3,5,4,2->1,4,5,3,2，此时交换后的依然是递增序列。
3. 新的递增序列逆序，即：1,4,5,3,2 => 1,4,2,3,5

```

1 from typing import List
2
3
4 def nextPermutation(nums: List[int]) -> None:
5     i = len(nums) - 2
6     while i >= 0 and nums[i] >= nums[i + 1]:
7         i -= 1
8     if i >= 0:
9         j = len(nums) - 1
10        while j >= 0 and nums[i] >= nums[j]:
11            j -= 1
12
13        nums[i], nums[j] = nums[j], nums[i]
14
15    left, right = i + 1, len(nums) - 1
16    while left < right:
17        nums[left], nums[right] = nums[right], nums[left]
18        left += 1
19        right -= 1
20
21
22 # =====
23 # n = int(input())
24 # m = int(input())
25 # arr = list(map(int, input().split()))
26 # for k in range(m):
27 #     nextPermutation(arr)
28 # print(*arr)
29 # =====
30
31 m = int(input())
32 for _ in range(m):
33     n, k = map(int, input().split())
34     a = list(map(int, input().split()))
35
36     for _ in range(k):
37         nextPermutation(a)
38

```

1.2 思路二：康托展开

康托展开 (Cantor Expansion) 来解决问题。康托展开是一种将排列映射到唯一整数的方法，可以用来快速找到某个排列的字典序编号，以及根据编号恢复排列。

```

1 # 2022fall-cs101, 陈勃宇
2 # cantor expansion
3
4 aa = [1]
5 c = 1
6 for i in range(1,1025):
7     c = c * i
8     aa.append(c)
9
10 for _ in range(int(input())):
11     n,k = map(int,input().split())
12     *cc, = map(int,input().split())
13     *bb, = range(1,n + 1)
14
15     d = 0
16     l = n - 1
17     for j in cc:
18         d = d + bb.index(j) * aa[l]
19         bb.remove(j)
20         l -= 1
21
22     d += k
23     while d >= aa[n]:
24         d -= aa[n]
25
26     dd = []
27     *bb, = range(1,n + 1)
28     for p in range(n - 1,-1,-1):
29         t = d // aa[p]
30         dd.append(bb[t])
31         del(bb[t])
32         d -= t * aa[p]
33 print(*dd)

```

程序中用到了remove, pop操作, gpt提醒是否用OrderedDict能优化。

```

1 # 计算阶乘
2 aa = [1]
3 for i in range(1, n+1):
4     aa.append(aa[-1] * i)
5
6 # 初始化 bb 和 pos (bb中每个元素的位置)
7 pos = [0] * (n+1)
8 bb = list(OrderedDict.fromkeys(range(1, n+1)))

```

示例：康托展开逆运算(cantor 2)

给出一个数N，再给出N的全排列的某一个排列的次序数，输出该排列。

Input

第1行为一个数 $N (N \leq 9)$ ，第2行为N的全排列的某一个排列的次序数。

Output

一行字符串，即该排列。

Sample in

3

1

Sample out

123

思路：可以用康托展开的逆运算来求解。假设已有{1,2,3,4,5}的全排列，并且已经从小到大排序完毕，现要找出第96个数的排列是什么，则康托展开逆运算的具体计算过程如下：

首先用 96-1 得到 95；

用 95 去除 4! 得到 3 余 23，商为 3 表示有 3 个数比它小，则该数是 4，所以第 1 位是 4；

用 23 去除 3! 得到 3 余 5，商为 3，表示有 3 个数比它小，即该数是 4，但 4 前面已经出现过了，所以第 2 位是 5；

用 5 去除 2! 得到 2 余 1，商为 2，表示有 2 个数比它小，即该数是 3，所以第 3 位是 3；

用 1 去除 1! 得到 1 余 0，表示有 1 个数比它小，即该数是 2，所以第 4 位是 2；

最后一个数只能是 1。

所以这个排列是 4 5 3 2 1。

又如找出第 16 个数的排列的计算过程如下：

首先用 16-1 得到 15；

用 15 去除 4! 得到 0 余 15，表示有 0 个数比它小，即该数是 1，第 1 位是 1；

用 15 去除 3! 得到 2 余 3，表示有 2 个数比它小，即该数是 3，但由于 1 已经在之前出现过了，所以第 2 位是 4（因为 1 在之前出现过了，所以实际上比 4 小的数是 2）；

用3去除 $2!$ 得到1余1，表示有1个数比它小，即该数是2，但由于1已经在之前出现过了，所以第3位是3（因为1在之前出现过了，所以实际上比3小的数是1）；

用1去除 $1!$ 得到1余0，表示有1个数比它小，即该数是2，但由于1、3、4已经在之前出现过了，所以第4位是5（因为1、3、4在之前出现过了，所以实际上比5小的数是1）。

最后一个数只能是2，所以这个数是14352。

参考代码如下。

```
1 import math
2
3 def cantor(m, n):
4     fac = [1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880]      # 预处理求出阶乘的值
5     hash = [0] * 10
6
7     num = 0
8     m -= 1
9     for i in range(n - 1, 0, -1):
10        used = 0
11        digit = m // fac[i] + 1                                # 计算有几个数比它小后加1
12        m %= fac[i]                                         # 更新m
13        for j in range(1, used + digit + 1):                  # 查找之前有哪些数已被用过
14            if hash[j]:
15                used += 1
16            num += (used + digit) * math.pow(10, i)
17            hash[used + digit] = 1                            # 标记该数被使用过
18
19        for i in range(1, n + 1):                                # 取出最后的未被使用的数
20            if hash[i] == 0:
21                return int(num + i)
22
23    return -1
24
25 num, n = map(int, input().split())
26 perm = cantor(n, num)
27 print(''.join(str(perm)))
```

leetcode.cn/playground/new/empty/

力扣 学习 题库 竞赛 讨论 求职 商店

执行代码 未命名 保存 Python3 输出: 完成

```
1 import math
2
3 def cantor(m, n):
4     fac = [1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880]    # 预处理求出阶乘的值
5     hash = [0] * 10
6
7     num = 0
8     m -= 1
9     for i in range(n - 1, 0, -1):
10         used = 0
11         digit = m // fac[i] + 1                         # 计算有几个数比它小后加1
12         m %= fac[i]                                     # 更新m
13         for j in range(1, used + digit + 1):
14             if hash[j]:
15                 used += 1
16             num += (used + digit) * math.pow(10, i)        # 标记该数被使用过
17             hash[used + digit] = 1
18
19         for i in range(1, n + 1):                         # 取出最后的未被使用的数
20             if hash[i] == 0:
21                 return int(num + i)
22
23     return -1
24
25 num, n = map(int, input().split())
26 perm = cantor(n, num)
27 print(''.join(str(perm)))
28
```

执行完成, 耗时: 48 ms
4 5 3 2 1

stdin 5 96

```
1 //康托展开逆运算
2 #include <bits/stdc++.h>
3 using namespace std;
4
5 int fac[10] = {1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880};    //预处理求出阶乘的值
6 int Hash[10];
7
8 int Cantor(int m, int n)
9 {
10     int num = 0;
11     int used, digit;
12     m--;
13     for (int i = n - 1; i > 0; i--)
14     {
15         used = 0;
16         digit = m / fac[i] + 1;                      //计算有几个数比它小后加1
17         m %= fac[i];                                //更新m
```

```
18     for(int j=1; j<=used+digit; j++)           //查找之前有哪些数已被用过
19         if(Hash[j])
20             used++;
21         num += (used+digit)*pow(10,i);
22         Hash[used + digit]=1;                   //标记该数被使用过
23     }
24     for(int i=1; i<=n; i++)                  //取出最后的未被使用的数
25         if(Hash[i] == 0)
26             return num+i;
27
28     return -1;
29 }
30
31 int main()
32 {
33     int num,n;
34     cin >> num >> n;
35     printf("%d\n",Cantor(n,num));
36     return 0;
37 }
```

附录A 数据结构：线段树和树状数组

理解时间复杂度 $O(1)$ 和 $O(n)$ 权衡处理方法，有的题目 $O(n^2)$ 算法超时，需要把时间复杂度降到 $O(n \log n)$ 才能 AC。

例如：27018:康托展开，<http://cs101.openjudge.cn/practice/27018/>

线段树（Segment Tree）和树状数组（Binary Indexed Tree）的区别和联系：

- 1) 时间复杂度相同，但是树状数组的常数优于线段树。
- 2) 树状数组的作用被线段树完全涵盖，凡是能够使用树状数组解决的问题，使用线段树一定可以解决，但是线段树能够解决的问题树状数组未必能够解决。
- 3) 树状数组的代码量比线段树小很多。

Segment Tree and Its Applications

<https://www.baeldung.com/cs/segment-trees#:~:text=The%20segment%20tree%20is%20a,structure%20such%20as%20an%20array.>

The segment tree is a type of data structure from computational geometry. [Bentley](#) proposed this well-known technique in 1977. A segment tree is essentially a binary tree in whose nodes we store the information about the segments of a linear data structure such as an array.

区间树是一种来自计算几何的数据结构。Bentley 在 1977 年提出了这一著名的技巧。区间树本质上是一棵二叉树，在其节点中存储了关于线性数据结构（如数组）的区段信息。

Fenwick tree

https://en.wikipedia.org/wiki/Fenwick_tree#:~:text=A%20Fenwick%20tree%20or%20binary,in%20an%20array%20of%20values.&text=This%20structure%20was%20proposed%20by,further%20modification%20published%20in%201992.

A **Fenwick tree** or **binary indexed tree (BIT)** is a data structure that can efficiently update values and calculate [prefix sums](#) in an array of values.

This structure was proposed by Boris Ryabko in 1989 with a further modification published in 1992. It has subsequently become known under the name Fenwick tree after Peter Fenwick, who described this structure in his 1994 article.

Fenwick 树或二叉索引树 (BIT) 是一种数据结构，可以高效地更新数组中的值并计算前缀和。

这种结构由 Boris Ryabko 于 1989 年提出，并在 1992 年进行了进一步的修改。此后，这种结构以其在 1994 年的文章中描述它的 Peter Fenwick 的名字而广为人知，被称为 Fenwick 树。

A.1 Segment tree | Efficient implementation

<https://www.geeksforgeeks.org/segment-tree-efficient-implementation/>

Let us consider the following problem to understand Segment Trees without recursion.

We have an array $arr[0 \dots n - 1]$. We should be able to,

1. Find the sum of elements from index l to r where $0 \leq l \leq r \leq n - 1$
2. Change the value of a specified element of the array to a new value x . We need to do $arr[i] = x$ where $0 \leq i \leq n - 1$.

A **simple solution** is to run a loop from l to r and calculate the sum of elements in the given range. To update a value, simply do $arr[i] = x$. The first operation takes $O(n)$ time and the second operation takes $O(1)$ time.

简单解决方案 是从 l 到 r 运行一个循环，计算给定范围内的元素之和。要更新一个值，只需执行 $arr[i] = x$ 。第一个操作（查询）的时间复杂度为 $O(n)$ ，第二个操作（更新）的时间复杂度为 $O(1)$ 。

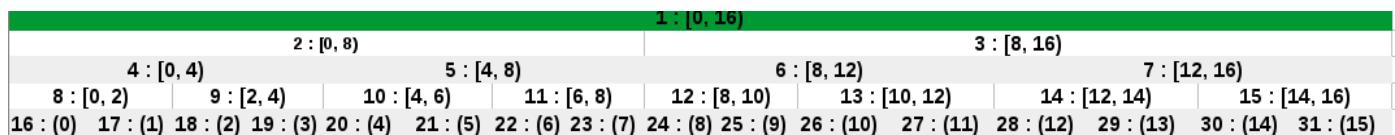
Another solution is to create another array and store the sum from start to i at the i th index in this array. The sum of a given range can now be calculated in $O(1)$ time, but the update operation takes $O(n)$ time now. This works well if the number of query operations is large and there are very few updates.

另一种解决方案 是创建另一个数组，并在该数组的第 i 个索引处存储从起始位置到 i 的元素之和。现在可以在 $O(1)$ 时间内计算给定范围的和，但更新操作现在需要 $O(n)$ 时间。如果查询操作的数量很大而更新操作很少，这种方法效果很好。

What if the number of queries and updates are equal? Can we perform both the operations in $O(\log n)$ time once given the array? We can use a [Segment Tree](#) to do both operations in $O(\log n)$ time. We have discussed the complete implementation of segment trees in our [previous](#) post. In this post, we will discuss the easier and yet efficient implementation of segment trees than in the previous post.

但如果查询和更新操作的数量相等呢？我们能否在给定数组的情况下，使两个操作都在 $O(\log n)$ 时间内完成？我们可以使用 线段树 来在 $O(\log n)$ 时间内完成这两个操作。我们在之前的帖子中详细讨论了线段树的完整实现。在这篇文章中，我们将讨论比之前更简单且高效的线段树实现方法。

Consider the array and segment tree as shown below: 叶子是数组值，非叶是和



You can see from the above image that the original array is at the bottom and is 0-indexed with 16 elements. The tree contains a total of 31 nodes where the leaf nodes or the elements of the original array start from node 16. So, we can easily construct a segment tree for this array using a 2^*N sized array where N is the number of elements in the original array. The leaf nodes will start from index N in this array and will go up to index $(2 * N - 1)$. Therefore, the element at index i in the original array will be at index $(i + N)$ in the segment tree array. Now to calculate the parents, we will start from the index $(N - 1)$ and move upward. For index i , the left child will be at $(2 * i)$ and the right child will be at $(2 * i + 1)$ index. So the values at nodes at $(2 * i)$ and $(2 * i + 1)$ are combined at i -th node to construct the tree.

从上图可以看出，原始数组位于底部，是 0 索引的，包含 16 个元素。树总共有 31 个节点，其中叶节点或原始数组的元素从节点 16 开始。因此，我们可以使用一个大小为 2^*N 的数组轻松构建这个数组的线段树，其中 N 是原始数组中的元素数量。叶节点将从该数组的索引 N 开始，一直到索引 $(2 * N - 1)$ 。因此，原始数组中索引 i 处的元素将在线段树数组中的索引 $(i + N)$ 处。现在，为了计算父节点，我们将从索引 $(N - 1)$ 开始向

上移动。对于索引 i , 左孩子将位于 $(2 * i)$ 索引处, 右孩子将位于 $(2 * i + 1)$ 索引处。因此, 节点 $(2 * i)$ 和 $(2 * i + 1)$ 处的值将在 i 索引处组合以构建树。

As you can see in the above figure, we can query in this tree in an interval $[L, R]$ with left index (L) included and right (R) excluded.

We will implement all of these multiplication and addition operations using bitwise operators.

如上图所示, 我们可以在区间 $[L, R]$ 中查询这棵树, 其中左索引 L 包含在内, 右索引 R 排除在外。

我们将使用位运算符实现所有的乘法和加法操作。

Let us have a look at the complete implementation:

```
1 # Python3 Code Addition
2
3 # limit for array size
4 N = 100000;
5
6 # Max size of tree
7 tree = [0] * (2 * N);
8
9 # function to build the tree
10 def build(arr) :
11
12     # insert leaf nodes in tree
13     for i in range(n) :
14         tree[n + i] = arr[i];
15
16     # build the tree by calculating parents
17     for i in range(n - 1, 0, -1) :
18         # tree[i] = tree[2*i] + tree[2*i+1]
19         tree[i] = tree[i << 1] + tree[i << 1 | 1];
20
21 # function to update a tree node
22 def updateTreeNode(p, value) :
23
24     # set value at position p
25     tree[p + n] = value;
26     p = p + n;
27
28     # move upward and update parents
29     i = p;
30
31     while i > 1 :
32
33         tree[i >> 1] = tree[i] + tree[i ^ 1];
34         i >>= 1;
35
36 # function to get sum on interval [l, r)
37 def query(l, r) :
38
39     res = 0;
```

```

41 # loop to find the sum in the range
42 l += n;
43 r += n;
44
45 while l < r :
46
47     if (l & 1) :
48         res += tree[l];
49         l += 1
50
51     if (r & 1) :
52         r -= 1;
53         res += tree[r];
54
55     l >>= 1;
56     r >>= 1
57
58 return res;
59
60 if __name__ == "__main__":
61
62     a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12];
63
64     n = len(a);
65
66     build(a);
67
68     # print the sum in range(1,2) index-based
69     print(query(1, 3));
70
71     # modify element at 2nd index
72     updateTreeNode(2, 1);
73
74     # print the sum in range(1,2) index-based
75     print(query(1, 3));
76

```

Output:

1	5
2	3

Yes! That is all. The complete implementation of the segment tree includes the query and update functions. Let us now understand how each of the functions works:

1. The picture makes it clear that the leaf nodes are stored at $i+n$, so we can clearly insert all leaf nodes directly.

图片清楚地表明叶节点存储在 $i+n$ 的位置，因此我们可以直接明确地插入所有叶节点。

2. The next step is to build the tree and it takes $O(n)$ time. The parent always has its less index than its children, so we just process all the nodes in decreasing order, calculating the value of the parent node. If the code inside the build function to calculate parents seems confusing, then you can see this code. It is equivalent to that inside the build function.

下一步是构建树，这需要 $O(n)$ 的时间。父节点的索引总是小于其子节点的索引，所以我们只需按递减顺序处理所有节点，计算父节点的值。如果构建函数中用于计算父节点的代码看起来令人困惑，那么你可以参考这段代码。它与构建函数内部的代码等效。

```
tree[i] = tree[2*i] + tree[2*i+1]
```

3. Updating a value at any position is also simple and the time taken will be proportional to the height (“高度”这个概念，其实就是从下往上度量，树这种数据结构的高度是从最底层开始计数，并且计数的起点是0) of the tree. We only update values in the parents of the given node which is being changed. So to get the parent, we just go up to the parent node, which is $p/2$ or $p>>1$, for node p . p^1 turns $(2*i)$ to $(2*i + 1)$ and vice versa to get the second child of p .

在任意位置更新一个值也非常简单，所需时间将与树的高度成正比。我们只更新给定节点（即正在更改的节点）的父节点中的值。为了得到父节点，我们只需向上移动到节点p的父节点，该父节点为 $p/2$ 或 $p>>1$ 。 p^1 将 $(2*i)$ 转换为 $(2*i + 1)$ 反之亦然，以获得p的第二个子节点。

4. Computing the sum also works in $O(Logn)$ time. If we work through an interval of [3,11), we need to calculate only for nodes 19,26,12, and 5 in that order. 要演示这个索引上行的求和过程，前面程序数组是12个元素，图示是16个元素，需要稍作修改。增加了print输出，便于调试。

```

37 # function to get sum on interval [l, r)
38 def query(l, r) :
39
40     res = 0;
41
42     # loop to find the sum in the range
43     l += n;
44     r += n;
45
46     while l < r :
47
48         if (l & 1) :
49             print(f'left index = {l}, value = {tree[l]}')
50             res += tree[l];
51             l += 1
52
53         if (r & 1) :
54             r -= 1;
55             print(f'right index = {r}, value = {tree[r]}')
56             res += tree[r];
57
58         l >>= 1;
59         r >>= 1
60
61     return res;
62
63 if __name__ == "__main__":
64
65     #a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12];
66     a = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15];
67
68     n = len(a);
69
70     build(a);
71
72     # print the sum in range(3,11) index-based
73     print(query(3, 11))
74

```

```
In [19]: runfile('/Users/hfyan/u
left index = 19, value = 3
right index = 26, value = 10
right index = 12, value = 17
left index = 5, value = 22
52

In [20]:
```

The idea behind the query function is whether we should include an element in the sum or whether we should include its parent. Let's look at the image once again for proper understanding.

1 : [0, 16]							
2 : [0, 8]				3 : [8, 16]			
4 : [0, 4)		5 : [4, 8)		6 : [8, 12)		7 : [12, 16)	
8 : [0, 2)	9 : [2, 4)	10 : [4, 6)	11 : [6, 8)	12 : [8, 10)	13 : [10, 12)	14 : [12, 14)	15 : [14, 16)
16 : (0)	17 : (1)	18 : (2)	19 : (3)	20 : (4)	21 : (5)	22 : (6)	23 : (7)
24 : (8)	25 : (9)	26 : (10)	27 : (11)	28 : (12)	29 : (13)	30 : (14)	31 : (15)

Consider that L is the left border of an interval and R is the right border of the interval $[L, R]$. It is clear from the image that if L is odd, then it means that it is the right child of its parent and our interval includes only L and not the parent. So we will simply include this node to sum and move to the parent of its next node by doing $L = (L+1)/2$. Now, if L is even, then it is the left child of its parent and the interval includes its parent also unless the right borders interfere. Similar conditions are applied to the right border also for faster computation. We will stop this iteration once the left and right borders meet.

假设 L 是一个区间的左边界，而 R 是区间 $[L, R]$ 的右边界。从图中可以明显看出，如果 L 是奇数，这意味着它是其父节点的右孩子，并且我们的区间仅包含 L 而不包括其父节点。因此，我们将简单地把这个节点加到总和中，并通过执行 $L = (L+1)/2$ 移动到下一个节点的父节点。现在，如果 L 是偶数，那么它是其父节点的左孩子，除非右边界干涉，否则区间也包括其父节点。对于右边界也有类似的条件，以便更快地计算。一旦左右边界相遇，我们就会停止这次迭代。

The theoretical time complexities of both previous implementation and this implementation is the same, but practically, it is found to be much more efficient as there are no recursive calls. We simply iterate over the elements that we need. Also, this is very easy to implement.

这两种实现的理论时间复杂度是相同的，但在实际应用中，后者被发现要高效得多，因为没有递归调用。我们只是迭代我们需要的元素。此外，这种方法非常容易实现。

Time Complexities:

- Tree Construction: $O(n)$
- Query in Range: $O(\log n)$
- Updating an element: $O(\log n)$.

Auxiliary Space: $O(2^N)$

示例1364A: A. XXXXX

brute force/data structures/number theory/two pointers, 1200, <https://codeforces.com/problemset/problem/1364/A>

Ehab loves number theory, but for some reason he hates the number x . Given an array a , find the length of its longest subarray such that the sum of its elements **isn't** divisible by x , or determine that such subarray doesn't exist.

An array a is a subarray of an array b if a can be obtained from b by deletion of several (possibly, zero or all) elements from the beginning and several (possibly, zero or all) elements from the end.

Input

The first line contains an integer t ($1 \leq t \leq 5$) — the number of test cases you need to solve. The description of the test cases follows.

The first line of each test case contains 2 integers n and x ($1 \leq n \leq 10^5$, $1 \leq x \leq 10^4$) — the number of elements in the array a and the number that Ehab hates.

The second line contains n space-separated integers a_1, a_2, \dots, a_n ($0 \leq a_i \leq 10^4$) — the elements of the array a .

Output

For each testcase, print the length of the longest subarray whose sum isn't divisible by x . If there's no such subarray, print -1 .

Example

input

```
1 3
2 3 3
3 1 2 3
4 3 4
5 1 2 3
6 2 2
7 0 6
```

output

```
1 2
2 3
3 -1
```

Note

In the first test case, the subarray [2,3] has sum of elements 5, which isn't divisible by 3.

In the second test case, the sum of elements of the whole array is 6, which isn't divisible by 4.

In the third test case, all subarrays have an even sum, so the answer is -1 .

Pypy3 可以AC。使用tree segment, 时间复杂度是 $O(n * \log n)$

```
1 # CF 1364A
2
3 # def prefix_sum(nums):
4 #     prefix = []
5 #     total = 0
6 #     for num in nums:
7 #         total += num
8 #         prefix.append(total)
9 #     return prefix
```

```

10
11 # def suffix_sum(nums):
12 #     suffix = []
13 #     total = 0
14 #     # 首先将列表反转
15 #     reversed_nums = nums[::-1]
16 #     for num in reversed_nums:
17 #         total += num
18 #         suffix.append(total)
19 #     # 将结果反转回来
20 #     suffix.reverse()
21 #     return suffix
22
23
24 t = int(input())
25 ans = []
26 for _ in range(t):
27     n, x = map(int, input().split())
28     a = [int(i) for i in input().split()]
29
30
31 # Segment tree | Efficient implementation
32 # https://www.geeksforgeeks.org/segment-tree-efficient-implementation/
33
34     # Max size of tree
35     tree = [0] * (2 * n);
36
37     def build(arr) :
38
39         # insert leaf nodes in tree
40         for i in range(n) :
41             tree[n + i] = arr[i];
42
43         # build the tree by calculating parents
44         for i in range(n - 1, 0, -1) :
45             tree[i] = tree[i << 1] + tree[i << 1 | 1];
46
47         # function to update a tree node
48         def updateTreeNode(p, value) :
49
50             # set value at position p
51             tree[p + n] = value;
52             p = p + n;
53
54             # move upward and update parents
55             i = p;
56
57             while i > 1 :
58
59                 tree[i >> 1] = tree[i] + tree[i ^ 1];
60                 i >>= 1;
61

```

```

62     # function to get sum on interval [l, r)
63     def query(l, r) :
64
65         res = 0;
66
67         # loop to find the sum in the range
68         l += n;
69         r += n;
70
71         while l < r :
72
73             if (l & 1) :
74                 res += tree[l];
75                 l += 1
76
77             if (r & 1) :
78                 r -= 1;
79                 res += tree[r];
80
81             l >>= 1;
82             r >>= 1
83
84         return res;
85 #aprefix_sum = prefix_sum(a)
86 #asuffix_sum = suffix_sum(a)
87
88 build([i%x for i in a]);
89
90 left = 0
91 right = n - 1
92 if right == 0:
93     if a[0] % x !=0:
94         print(1)
95     else:
96         print(-1)
97     continue
98
99 leftmax = 0
100 rightmax = 0
101 while left != right:
102     #total = asuffix_sum[left]
103     total = query(left, right+1)
104     if total % x != 0:
105         leftmax = right - left + 1
106         break
107     else:
108         left += 1
109
110 left = 0
111 right = n - 1
112 while left != right:
113     #total = aprefix_sum[right]

```

```

114     total = query(left, right+1)
115     if total % x != 0:
116         rightmax = right - left + 1
117         break
118     else:
119         right -= 1
120
121     if leftmax == 0 and rightmax == 0:
122         #print(-1)
123         ans.append(-1)
124     else:
125         #print(max(leftmax, rightmax))
126         ans.append(max(leftmax, rightmax))
127
128 print('\n'.join(map(str,ans)))

```

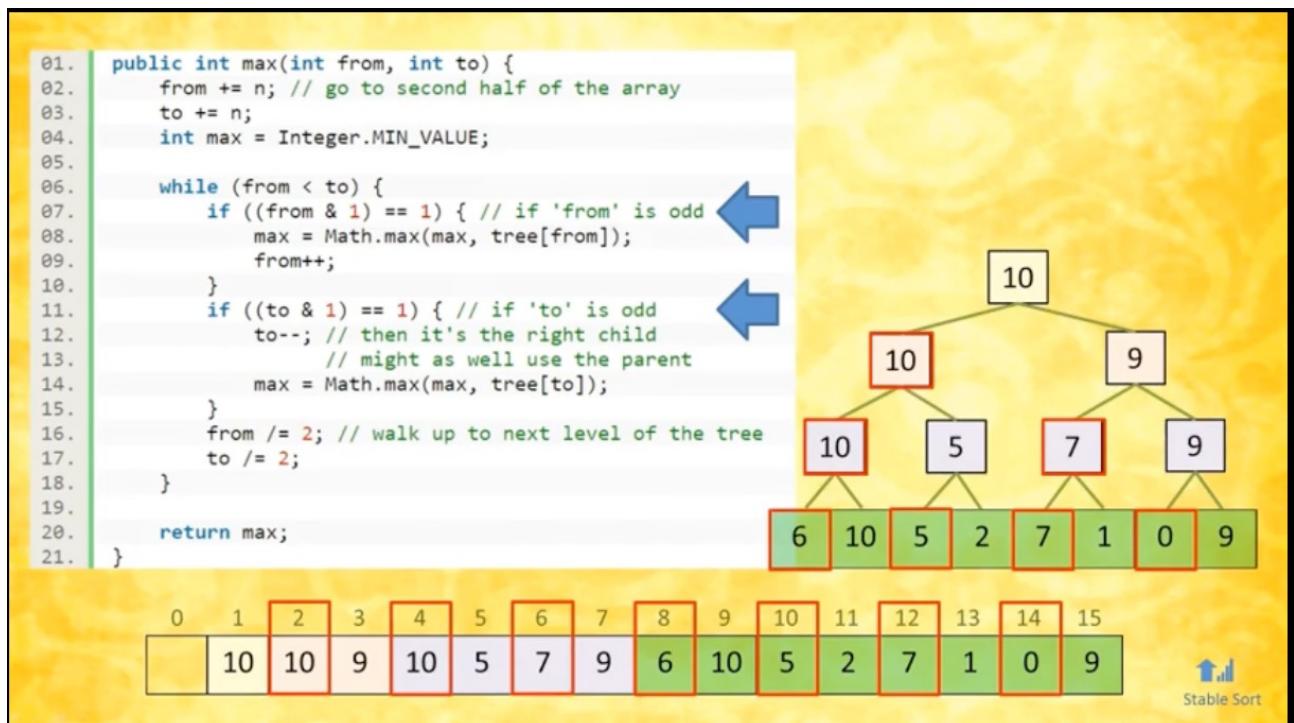
如果用sum求和， $O(n^2)$ ，pypy3也会在test3 超时。

Benefits of segment tree usage

<https://www.geeksforgeeks.org/segment-tree-sum-of-given-range/>

Range Queries: One of the main use cases of segment trees is to perform range queries on an array in an efficient manner. The query function in the segment tree can return the minimum, maximum, sum, or any other aggregation of elements within a specified range in the array in $O(\log n)$ time.

区间查询：线段树的主要用途之一是以高效的方式对数组进行区间查询。线段树中的查询函数可以在 $O(\log n)$ 时间内返回指定区间内元素的最小值、最大值、和或其他聚合结果。



假设根节点下标从0开始，左子节点 = 2*父节点+1，右子节点 = 2*父节点+2

二叉树的父子节点位置关系，<https://zhuanlan.zhihu.com/p/339763580>

```

1 class SegmentTree:
2     def __init__(self, array):
3         self.size = len(array)
4         self.tree = [0] * (4 * self.size)
5         self.build_tree(array, 0, 0, self.size - 1)
6
7     def build_tree(self, array, tree_index, left, right):
8         if left == right:
9             self.tree[tree_index] = array[left]
10            return
11        mid = (left + right) // 2
12        self.build_tree(array, 2 * tree_index + 1, left, mid)
13        self.build_tree(array, 2 * tree_index + 2, mid + 1, right)
14        self.tree[tree_index] = min(self.tree[2 * tree_index + 1], self.tree[2 * tree_index + 2])
15
16    def query(self, tree_index, left, right, query_left, query_right):
17        if query_left <= left and right <= query_right:
18            return self.tree[tree_index]
19        mid = (left + right) // 2
20        min_value = float('inf')
21        if query_left <= mid:
22            min_value = min(min_value, self.query(2 * tree_index + 1, left, mid,
23            query_left, query_right))
23        if query_right > mid:
24            min_value = min(min_value, self.query(2 * tree_index + 2, mid + 1, right,
25            query_left, query_right))

```

```

25     return min_value
26
27     def query_range(self, left, right):
28         return self.query(0, 0, self.size - 1, left, right)
29
30
31 if __name__ == '__main__':
32     array = [1, 3, 2, 5, 4, 6]
33     st = SegmentTree(array)
34     print(st.query_range(1, 5)) # 2
35

```

如果要返回区间最大值，只需要修改第14、20、22、24行程序为求最大相应代码

```

1      #self.tree[tree_index] = min(self.tree[2 * tree_index + 1], self.tree[2 *
tree_index + 2])
2          self.tree[tree_index] = max(self.tree[2 * tree_index + 1], self.tree[2 *
tree_index + 2])
3  ...
4      #min_value = float('inf')
5      min_value = -float('inf')
6      if query_left <= mid:
7          #min_value = min(min_value, self.query(2 * tree_index + 1, left, mid,
query_left, query_right))
8          min_value = max(min_value, self.query(2 * tree_index + 1, left, mid,
query_left, query_right))
9          if query_right > mid:
10              #min_value = min(min_value, self.query(2 * tree_index + 2, mid + 1,
right, query_left, query_right))
11              min_value = max(min_value, self.query(2 * tree_index + 2, mid + 1, right,
query_left, query_right))
12      return min_value
13  ...
14  print(st.query_range(1, 5)) # 6
15

```

如果要返回区间求和，只需要修改第14、20、22、24行程序为求和代码。

A.2 树状数组 (Binary Indexed Tree)

树状数组或二叉索引树（英语：Binary Indexed Tree），又以其发明者命名为Fenwick树，最早由Peter M. Fenwick于1994年以A New Data Structure for Cumulative Frequency Tables为题发表。其初衷是解决数据压缩里的累积频率（Cumulative Frequency）的计算问题，现多用于高效计算数列的前缀和，区间和。

Binary Indexed Tree or Fenwick Tree

<https://www.geeksforgeeks.org/binary-indexed-tree-or-fenwick-tree-2/>

Let us consider the following problem to understand Binary Indexed Tree.

We have an array $arr[0 \dots n - 1]$. We would like to

1 Compute the sum of the first i elements.

2 Modify the value of a specified element of the array $arr[i] = x$ where $0 \leq i \leq n - 1$.

A **simple solution** is to run a loop from 0 to $i-1$ and calculate the sum of the elements. To update a value, simply do $arr[i] = x$. The first operation takes $O(n)$ time and the second operation takes $O(1)$ time. Another simple solution is to create an extra array and store the sum of the first i -th elements at the i -th index in this new array. The sum of a given range can now be calculated in $O(1)$ time, but the update operation takes $O(n)$ time now. This works well if there are a large number of query operations but a very few number of update operations.

Could we perform both the query and update operations in $O(\log n)$ time?

One efficient solution is to use [Segment Tree](#) that performs both operations in $O(\log n)$ time.

An alternative solution is Binary Indexed Tree, which also achieves $O(\log n)$ time complexity for both operations. Compared with Segment Tree, Binary Indexed Tree requires less space and is easier to implement.

让我们考虑以下问题来理解二叉索引树 (Binary Indexed Tree, BIT) :

我们有一个数组 $arr[0 \dots n - 1]$ 。我们希望实现两个操作:

1. 计算前 i 个元素的和。
2. 修改数组中指定位置的值, 即设置 $arr[i] = x$, 其中 $0 \leq i \leq n - 1$ 。

一个简单的解决方案是从0到 $i-1$ 遍历并计算这些元素的总和。要更新一个值, 只需执行 $arr[i] = x$ 。第一个操作的时间复杂度为 $O(n)$, 而第二个操作的时间复杂度为 $O(1)$ 。另一种简单的解决方案是创建一个额外的数组, 并在这个新数组的第 i 个位置存储前 i 个元素的总和。这样, 给定范围的和可以在 $O(1)$ 时间内计算出来, 但是更新操作现在需要 $O(n)$ 时间。当查询操作非常多而更新操作非常少时, 这种方法表现良好。

我们能否在 $O(\log n)$ 时间内同时完成查询和更新操作呢?

一种高效的解决方案是使用段树 (Segment Tree), 它能够在 $O(\log n)$ 时间内完成这两个操作。

另一种解决方案是二叉索引树 (Binary Indexed Tree, 也称作 Fenwick Tree), 同样能够以 $O(\log n)$ 的时间复杂度完成查询和更新操作。与段树相比, 二叉索引树所需的空间更少, 且实现起来更加简单。

Idea: split the running total array into 2 smaller arrays, T1[] and T2[]

A =	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	5	2	9	-3	5	20	10	-7	2	3	-4	0	-2	15	5

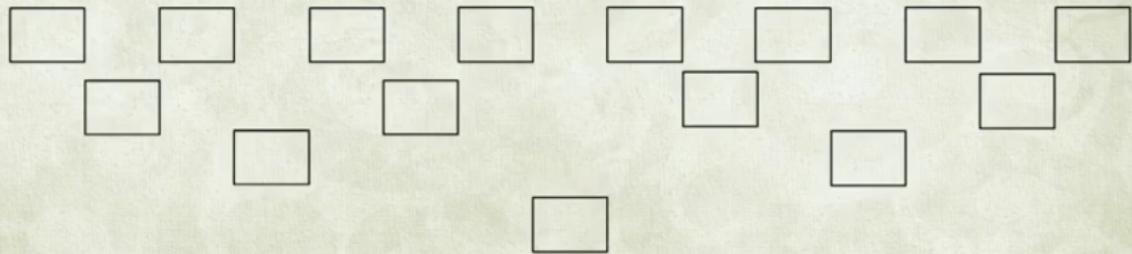
T1 =	5	7	16	13	18	38	48	41
------	---	---	----	----	----	----	----	----

T2 =	0	1	2	3	4	5	6
	2	5	1	1	-1	14	19

$$\text{sum}(5, 10) = T1[7] - T1[4] + T2[2] = 41 - 18 + 1 = 24$$



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
5	2	9	-3	5	20	10	-7	2	3	-4	0	-2	15	5



A	
5	1 00001
2	2 00010
9	3 00011
-3	4 00100
5	5 00101
20	6 00110
10	7 00111
-7	8 01000
2	9 01001
3	10 01010
-4	11 01011
0	12 01100
-2	13 01101
15	14 01110
5	15 01111

A New Data Structure for Cumulative Frequency Tables

peter m. fenwick

Department of Computer Science, University of Auckland, Private Bag 92019, Auckland,
New Zealand (email: p-fenwick@cs.auckland.ac.nz)

SUMMARY

A new method (the ‘binary indexed tree’) is presented for maintaining the cumulative frequencies which are needed to support dynamic arithmetic data compression. It is based on a decomposition of the cumulative frequencies into portions which parallel the binary representation of the index of the table element (or symbol). The operations to traverse the data structure are based on the binary coding of the index. In comparison with previous methods, the binary indexed tree is faster, using more compact data and simpler code. The access time for all operations is either constant or proportional to the logarithm of the table size. In conjunction with the compact data structure, this makes the new method particularly suitable for large symbol alphabets.

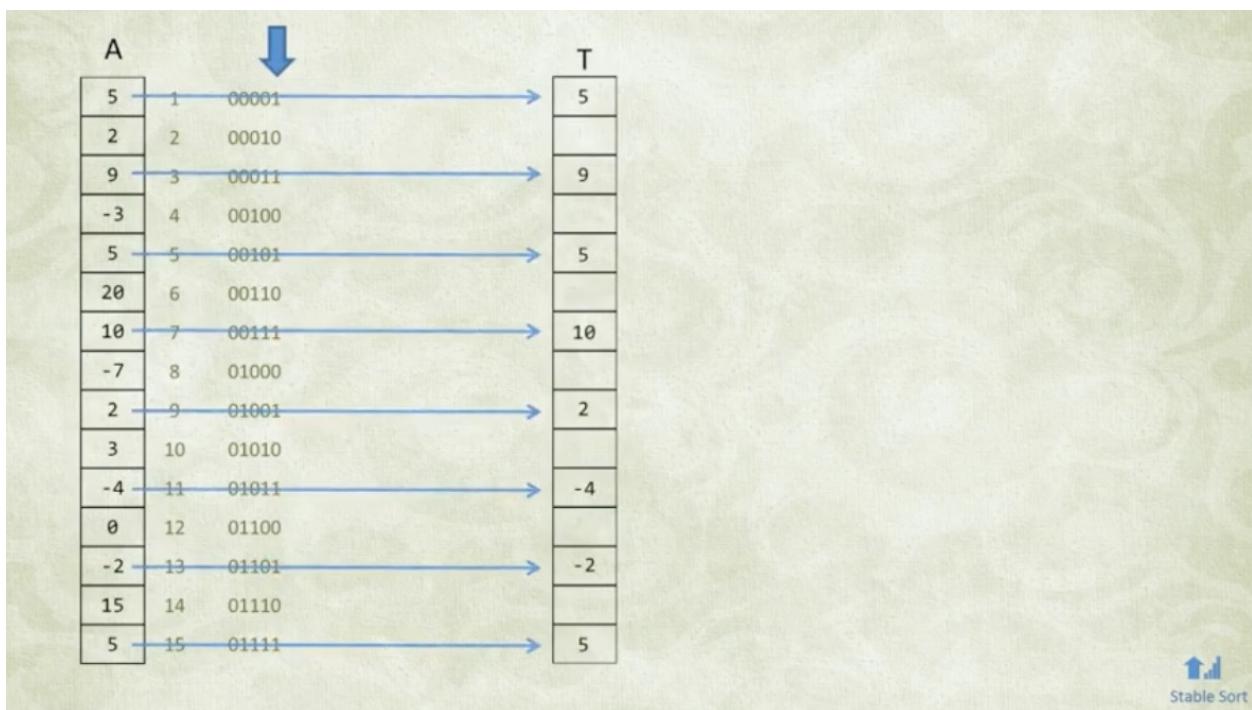


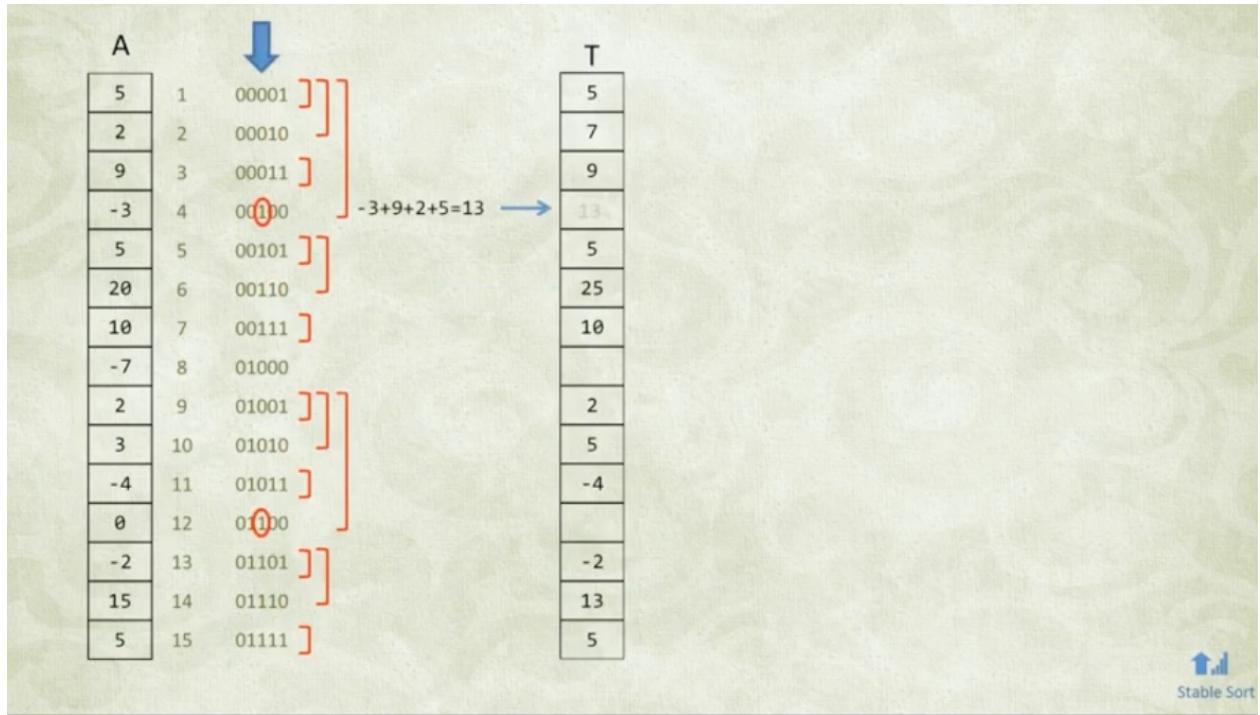
Representation

Binary Indexed Tree is represented as an array. Let the array be $\text{BITree}[]$. Each node of the Binary Indexed Tree stores the sum of some elements of the input array. The size of the Binary Indexed Tree is equal to the size of the input array, denoted as n . In the code below, we use a size of $n+1$ for ease of implementation.

表示方式

二叉索引树用数组形式表示。设该数组为 $\text{BITree}[]$ 。二叉索引树的每个节点存储了输入数组某些元素的和。二叉索引树的大小等于输入数组的大小，记为 n 。在下面的代码中，为了便于实现，我们使用 $n+1$ 的大小。





Construction

We initialize all the values in `BITree[]` as 0. Then we call `update()` for all the indexes, the `update()` operation is discussed below.

构建

我们首先将`BITree[]`中的所有值初始化为0。然后对所有的索引调用`update()`函数，下面将讨论`update()`操作的具体内容。

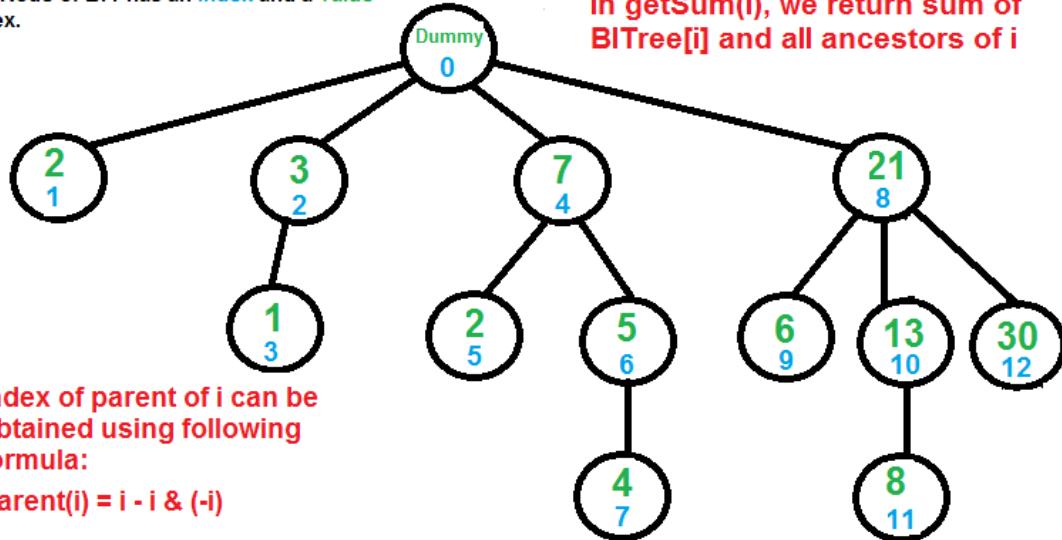
Operations

`getSum(x): Returns the sum of the sub-array arr[0,...,x]`

// Returns the sum of the sub-array `arr[0,...,x]` using `BITree[0..n]`, which is constructed from `arr[0..n-1]`

1. Initialize the output sum as 0, the current index as `x+1`.
2. Do following while the current index is greater than 0.
 - ...a) Add `BITree[index]` to sum
 - ...b) Go to the parent of `BITree[index]`. The parent can be obtained by removing the last set bit from the current index, i.e., `index = index - (index & (-index))`
3. Return sum.

Every Node of BIT has an **Index** and a **Value**
at index.



In `getSum(i)`, we return sum of `BITree[i]` and all ancestors of i

Index of parent of i can be obtained using following formula:

$$\text{parent}(i) = i - i \& (-i)$$

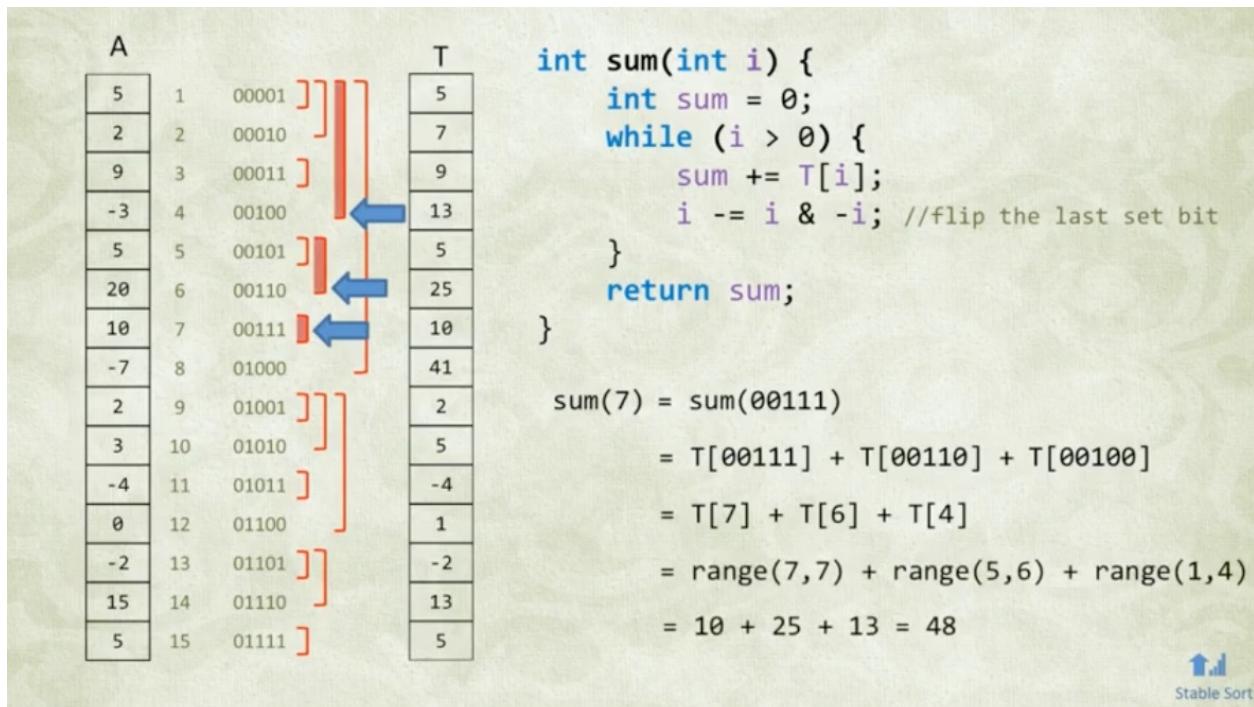
The above formula basically removes the last set bit from i. For example, if i = 12, then parent(i) is 8

Input Array: `arr[0..n-1]` = {2, 1, 1, 3, 2, 3, 4, 5, 6, 7, 8, 9}

BI Tree Array: `BITree[1..n]` = {2, 3, 1, 7, 2, 5, 4, 21, 6, 13, 8, 30}

View of Binary Indexed Tree to understand `getSum()` operation

`getsum(7)`



`getsum(8)`

A	T
5	5
2	7
9	9
-3	13
5	5
20	25
10	10
-7	41
2	2
3	5
-4	-4
0	1
-2	-2
15	13
5	5

Diagram illustrating the computation of sum(8) using a table T. Red brackets group elements into pairs: (5, 7), (9, 9), (-3, 13), (5, 5), (20, 25), (10, 10), (-7, 41), (2, 2), (3, 5), (-4, -4), (0, 1), (-2, -2), (15, 13), (5, 5). A blue arrow points from the value 8 to the pair (2, 2).

```

int sum(int i) {
    int sum = 0;
    while (i > 0) {
        sum += T[i];
        i -= i & -i; //flip the last set bit
    }
    return sum;
}

sum(8) = sum(01000)
        = T[01000]
        = T[8]
        = range(1,8)
        = 41

```

↑
Stable Sort

整数的二进制表示常用的方式之一是使用补码

补码是一种表示有符号整数的方法，它将负数的二进制表示转换为正数的二进制表示。补码的优势在于可以使用相同的算术运算规则来处理正数和负数，而不需要特殊的操作。

在补码表示中，最高位用于表示符号位，0表示正数，1表示负数。其他位表示数值部分。

具体将一个整数转换为补码的步骤如下：

1. 如果整数是正数，则补码等于二进制表示本身。
2. 如果整数是负数，则需要先将其绝对值转换为二进制，然后取反，最后加1。

例如，假设要将-5转换为补码：

1. 5的二进制表示为00000101。
2. 将其取反得到11111010。
3. 加1得到11111011，这就是-5的补码表示。

```

int sum(int i) {
    int sum = 0;
    while (i > 0) {
        sum += T[i];
        i -= i & -i; //flip the last set bit
    }
    return sum;
}

```

$$\begin{aligned}
7 &= (00111)_2 \\
-7 &= (11001)_2 \\
00111 \& 11001 &= 00001 \\
00111 - 00001 &= 00110 = (6)_{10}
\end{aligned}$$



The diagram above provides an example of how getSum() is working. Here are some important observations.

BITree[0] is a dummy node.

BITree[y] is the parent of BITree[x], if and only if y can be obtained by removing the last set bit from the binary representation of x, that is $y = x - (x \& (-x))$.

The child node BITree[x] of the node BITree[y] stores the sum of the elements between y(inclusive) and x(exclusive): arr[y,...,x].

上图提供了一个getSum()如何工作的例子。这里有一些重要的观察点：

- BITree[0]是一个虚拟节点。
- 如果仅通过从x的二进制表示中移除最后一个设置位（即最右边的1）可以得到y，则BITree[y]是BITree[x]的父节点，这可以表示为 $y = x - (x \& (-x))$ 。
- 节点BITree[y]的子节点BITree[x]存储了从y（包括y）到x（不包括x）之间元素的和：arr[y,...,x]。

update(x, val): Updates the Binary Indexed Tree (BIT) by performing arr[index] += val

// Note that the update(x, val) operation will not change arr[]. It only makes changes to BITree[]

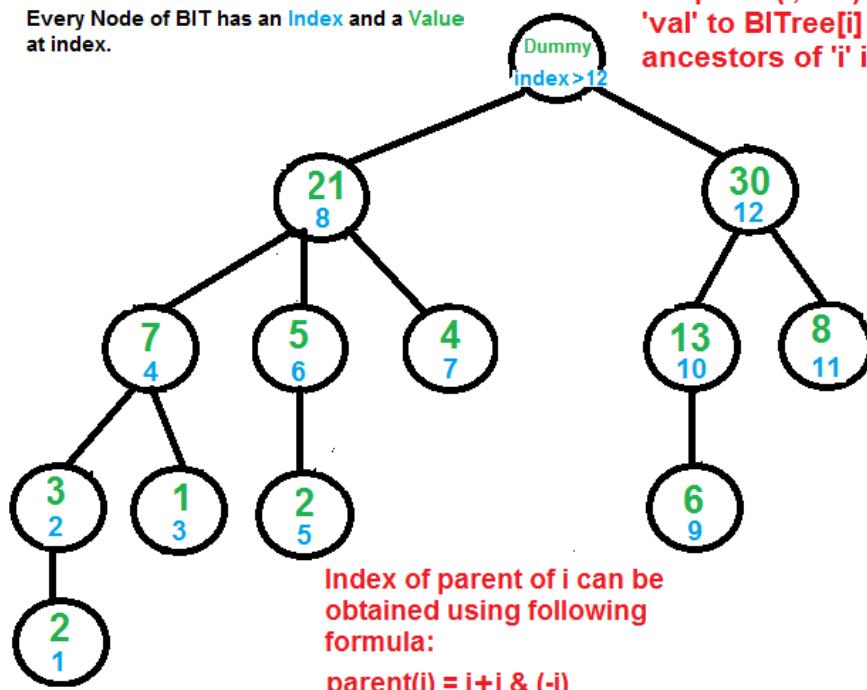
1. Initialize the current index as $x+1$.
2. Do the following while the current index is smaller than or equal to n.

...a) Add the val to BITree[index]

...b) Go to next element of BITree[index]. The next element can be obtained by incrementing the last set bit of the current index, i.e., $\text{index} = \text{index} + (\text{index} \& (-\text{index}))$

Every Node of BIT has an **Index** and a **Value** at index.

In update(*i*, *val*) we add '*val*' to **BITree[i]** and all ancestors of '*i*' in **BITree[]**

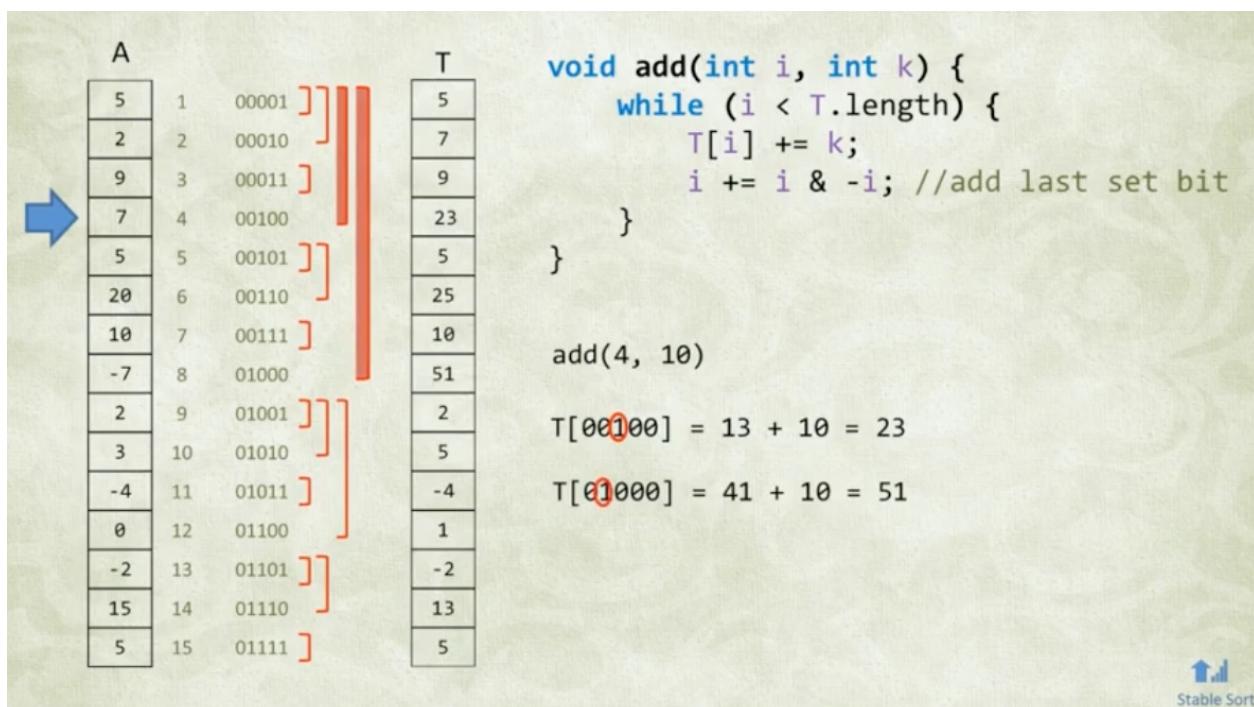


The above formula basically adds decimal value corresponding to the last set bit from *i*. For example, if *i* = 10 then parent(*i*) is 12

Contents of arr[] and BITree[] are same as above diagram for getSum()

View of Binary Indexed Tree to understand update() operation

update(4, 10)



The update function needs to make sure that all the BITree nodes which contain arr[i] within their ranges being updated. We loop over such nodes in the BITree by repeatedly adding the decimal number corresponding to the last set bit of the current index.

How does Binary Indexed Tree work?

The idea is based on the fact that all positive integers can be represented as the sum of powers of 2. For example 19 can be represented as $16 + 2 + 1$. Every node of the BITree stores the sum of n elements where n is a power of 2. For example, in the first diagram above (the diagram for getSum()), the sum of the first 12 elements can be obtained by the sum of the last 4 elements (from 9 to 12) plus the sum of 8 elements (from 1 to 8). The number of set bits in the binary representation of a number n is $O(\log n)$. Therefore, we traverse at-most $O(\log n)$ nodes in both getSum() and update() operations. The time complexity of the construction is $O(n \log n)$ as it calls update() for all n elements.

Implementation:

Following are the implementations of Binary Indexed Tree.

更新函数需要确保所有包含arr[i]在其范围内的BITree节点都被更新。我们通过不断向当前索引添加其最后一位设置位对应的十进制数，在BITree中循环遍历这些节点。

二叉索引树是如何工作的？

这个想法基于所有正整数都可以表示为2的幂的事实。例如，19可以表示为 $16 + 2 + 1$ 。BITree的每个节点都存储n个元素的和，其中n是2的幂。例如，在上面的第一个图（getSum()的图示）中，前12个元素的和可以通过最后4个元素（从9到12）的和加上前8个元素（从1到8）的和得到。一个数n的二进制表示中设置位的数量是 $O(\log n)$ 。因此，在getSum()和update()操作中，我们最多遍历 $O(\log n)$ 个节点。构建的时间复杂度为 $O(n \log n)$ ，因为它为所有n个元素调用了update()。

实现：

以下是二叉索引树的实现。

```
1 # Python implementation of Binary Indexed Tree
2
3 # Returns sum of arr[0..index]. This function assumes
4 # that the array is preprocessed and partial sums of
5 # array elements are stored in BITree[].
6 def getsum(BITTree,i):
7     s = 0 #initialize result
8
9     # index in BITree[] is 1 more than the index in arr[]
10    i = i+1
11
12    # Traverse ancestors of BITree[index]
13    while i > 0:
14
15        # Add current element of BITree to sum
16        s += BITTree[i]
17
18        # Move index to parent node in getSum View
19        i -= i & (-i)
20    return s
21
22 # Updates a node in Binary Index Tree (BITree) at given index
23 # in BITree. The given value 'val' is added to BITree[i] and
24 # all of its ancestors in tree.
25 def updatebit(BITTree , n , i ,v):
```

```

26
27     # index in BITree[] is 1 more than the index in arr[]
28     i += 1
29
30     # Traverse all ancestors and add 'val'
31     while i <= n:
32
33         # Add 'val' to current node of BI Tree
34         BITTree[i] += v
35
36         # Update index to that of parent in update View
37         i += i & (-i)
38
39
40     # Constructs and returns a Binary Indexed Tree for given
41     # array of size n.
42     def construct(arr, n):
43
44         # Create and initialize BITree[] as 0
45         BITTree = [0] * (n+1)
46
47         # Store the actual values in BITree[] using update()
48         for i in range(n):
49             updatebit(BITTree, n, i, arr[i])
50
51         # Uncomment below lines to see contents of BITree[]
52         #for i in range(1,n+1):
53         #    print BITTree[i],
54
55     return BITTree
56
57
58     # Driver code to test above methods
59     freq = [2, 1, 1, 3, 2, 3, 4, 5, 6, 7, 8, 9]
60     BITTree = construct(freq, len(freq))
61     print("Sum of elements in arr[0..5] is " + str(getsum(BITTree, 5)))
62     freq[3] += 6
63     updatebit(BITTree, len(freq), 3, 6)
64     print("Sum of elements in arr[0..5]"+
65           " after update is " + str(getsum(BITTree, 5)))
66
67     # This code is contributed by Raju Varshney

```

Output

```

1 | Sum of elements in arr[0..5] is 12
2 | Sum of elements in arr[0..5] after update is 18

```

Time Complexity: O(NLogN)

Auxiliary Space: O(N)

Can we extend the Binary Indexed Tree to computing the sum of a range in O(Logn) time?

Yes. $\text{rangeSum}(l, r) = \text{getSum}(r) - \text{getSum}(l-1)$.

Applications:

The implementation of the arithmetic coding algorithm. The development of the Binary Indexed Tree was primarily motivated by its application in this case. See [this](#) for more details.

Example Problems:

[Count inversions in an array | Set 3 \(Using BIT\)](#)

[Two Dimensional Binary Indexed Tree or Fenwick Tree](#)

[Counting Triangles in a Rectangular space using BIT](#)

References:

http://en.wikipedia.org/wiki/Fenwick_tree

<http://community.topcoder.com/tc?module=Static&d1=tutorials&d2=binaryIndexedTrees>

[力扣307] 线段树&树状数组, <https://zhuanlan.zhihu.com/p/126539401>

示例LeetCode307. 区域和检索 - 数组可修改

<https://leetcode.cn/problems/range-sum-query-mutable/>

给你一个数组 `nums`，请你完成两类查询。

1. 其中一类查询要求 **更新** 数组 `nums` 下标对应的值
2. 另一类查询要求返回数组 `nums` 中索引 `left` 和索引 `right` 之间（包含）的`nums`元素的 **和**，其中 `left <= right`

实现 `NumArray` 类：

- `NumArray(int[] nums)` 用整数数组 `nums` 初始化对象
- `void update(int index, int val)` 将 `nums[index]` 的值 **更新** 为 `val`
- `int sumRange(int left, int right)` 返回数组 `nums` 中索引 `left` 和索引 `right` 之间（包含）的 `nums`元素的 **和**（即，`nums[left] + nums[left + 1], ..., nums[right]`）

示例 1：

```
1 | 输入:  
2 | ["NumArray", "sumRange", "update", "sumRange"]  
3 | [[1, 3, 5]], [0, 2], [1, 2], [0, 2]  
4 | 输出:  
5 | [null, 9, null, 8]  
6 |  
7 | 解释:  
8 | NumArray numArray = new NumArray([1, 3, 5]);  
9 | numArray.sumRange(0, 2); // 返回 1 + 3 + 5 = 9  
10 | numArray.update(1, 2); // nums = [1,2,5]  
11 | numArray.sumRange(0, 2); // 返回 1 + 2 + 5 = 8
```

示例27018: 康托展开

<http://cs101.openjudge.cn/practice/27018/>

总时间限制: 3000ms 单个测试点时间限制: 2000ms 内存限制: 90112kB

描述

求 1~N 的一个给定全排列在所有 1~N 全排列中的排名。结果对 998244353 取模。

输入

第一行一个正整数 N。

第二行 N 个正整数，表示 1~N 的一种全排列。

输出

一行一个非负整数，表示答案对 998244353 取模的值。

样例输入

```
1 | Sample1 in:  
2 | 3  
3 | 2 1 3  
4 |  
5 | Sample1 output:  
6 | 3
```

样例输出

```
1 | Sample2 in:  
2 | 4  
3 | 1 2 4 3  
4 |  
5 | Sample2 output:  
6 | 2
```

提示: 对于 100% 数据, $1 \leq N \leq 1000000$ 。

来源: <https://www.luogu.com.cn/problem/P5367>

思路：容易想到的方法是把所有排列求出来后再进行排序，但事实上有更简单高效的算法来解决这个问题，那就是康托展开。

康托展开是一个全排列到一个自然数的双射，常用于构建特定哈希表时的空间压缩。康托展开的实质是计算当前排列在所有由小到大全排列中的次序编号，因此是可逆的。即由全排列可得到其次序编号（康托展开），由次序编号可以得到对应的第几个全排列（逆康托展开）。

康托展开的表达式为：

$$X = a_n \times (n-1)! + a_{n-1} \times (n-2)! + \dots + a_i \times (i-1)! + \dots + a_2 \times 1! + a_1 \times 0!$$

其中： X 为比当前排列小的全排列个数 ($X+1$ 即为当前排列的次序编号)； n 表示全排列表达式的字符串长度； a_i 表示原排列表达式中的第 i 位（由右往左数），前面（其右侧） $i-1$ 位数有多少个数的值比它小。

例如求 5 2 3 4 1 在 {1, 2, 3, 4, 5} 生成的排列中的次序可以按如下步骤计算。

从右往左数， i 是 5 时候，其右侧比 5 小的数有 1、2、3、4 这 4 个数，所以有 $4 \times 4!$ 。

是 2，比 2 小的数有 1 个数，所以有 $1 \times 3!$ 。

是 3，比 3 小的数有 1 个数，为 $1 \times 2!$ 。

是 4，比 4 小的数有 1 个数，为 $1 \times 1!$ 。

最后一位数右侧没有比它小的数，为 $0 \times 0! = 0$ 。

则 $4 \times 4! + 1 \times 3! + 1 \times 2! + 1 \times 1! = 105$ 。

这个 X 只是这个排列之前的排列数，而题目要求这个排列的位置，即 5 2 3 4 1 排在第 106 位。

同理，4 3 5 2 1 的排列数： $3 \times 4! + 2 \times 3! + 2 \times 2! + 1 \times 1! = 89$ ，即 4 3 5 2 1 排在第 90 位。

因为比 4 小的数有 3 个：3、2、1；比 3 小的数有 2 个：2、1；比 5 小的数有 2 个：2、1；比 2 小的数有 1 个：1。

参考代码如下。

```
1 MOD = 998244353          # Time Limit Exceeded, 内存7140KB, 时间18924ms
2 fac = [1]
3
4 def cantor_expand(a, n):
5     ans = 0
6
7     for i in range(1, n + 1):
8         count = 0
9         for j in range(i + 1, n + 1):
10            if a[j] < a[i]:
11                count += 1           # 计算有几个比他小的数
12            ans = (ans + (count * fac[n - i]) % MOD) % MOD
13    return ans + 1
14
15 a = [0]
16 N = int(input())      # 用大写N, 因为spyder的debug, 执行下一条指令的命令是 n/next。与变量n冲突。
17
18 for i in range(1, N + 1):
19     fac.append((fac[i - 1] * i) % MOD)      # 整数除法具有分配律
20
21 *perm, = map(int, input().split())
22 a.extend(perm)
23
```

```
24 | print(cantor_expand(a, N))
```

用C++也是超时

```
1 #include<iostream>           // Time Limit Exceeded, 内存960KB, 时间1986ms
2 using namespace std;
3
4 const long long MOD = 998244353;
5 long long fac[1000005]={1};
6
7 int cantor_expand (int a[],int n){
8     int i, j, count;
9     long long ans = 0 ;
10
11    for(i = 1; i <= n; i ++){
12        count = 0;
13        for(j = i + 1; j <= n; j ++){
14            if(a[j] < a[i]) count++;           // 计算有几个比它小的数
15        }
16        ans = (ans + (count * fac[n-i]) % MOD) % MOD;
17    }
18    return ans + 1;
19 }
20
21
22 int a[1000005];
23
24 int main()
25 {
26     int N;
27     //cin >> N;
28     scanf("%d", &N);
29     for (int i=1; i<=N; i++){
30         fac[i] = (fac[i-1]*i)%MOD;
31     }
32
33     for (int i=1; i<=N; i++)
34         //cin >> a[i];
35         scanf("%d",&a[i]);
36     cout << cantor_expand(a,N) << endl;
37     return 0;
38 }
```

优化

康托展开用 $O(n^2)$ 算法超时，需要把时间复杂度降到 $O(n \log n)$ 。“计算有几个比他小的数”，时间复杂度由 $O(n)$ 降到 $O(\log n)$ 。

树状数组 (Binary Indexed Tree)

实现树状数组的核心部分，包括了三个重要的操作：lowbit、修改和求和。

1. lowbit函数：`lowbit(x)` 是用来计算 `x` 的二进制表示中最低位的 `1` 所对应的值。它的运算规则是利用位运算 `(x & -x)` 来获取 `x` 的最低位 `1` 所对应的值。例如，`lowbit(6)` 的结果是 `2`，因为 `6` 的二进制表示为 `110`，最低位的 `1` 所对应的值是 `2`。

`-x` 是 `x` 的补码表示。

对于正整数 `x`，`-x` 的二进制表示是 `x` 的二进制表示取反后加 1。

`6` 的二进制表示为 `110`，取反得到 `001`，加 1 得到 `010`。

`-6` 的二进制表示为 `11111111111111111111111111010`（假设 32 位整数）。

`6 & -6` 的结果：

`110` 与 `111111111111111111111111010` 按位与运算，结果为 `010`，即 `2`。

2. update函数：这个函数用于修改树状数组中某个位置的值。参数 `x` 表示要修改的位置，参数 `y` 表示要增加/减少的值。函数使用一个循环将 `x` 的所有对应位置上的值都加上 `y`。具体的操作是首先将 `x` 位置上的值与 `y` 相加，然后通过 `lowbit` 函数找到 `x` 的下一个需要修改的位置，将该位置上的值也加上 `y`，然后继续找下一个位置，直到修改完所有需要修改的位置为止。这样就完成了数组的修改。

3. getsum函数：这个函数用于求解树状数组中某个范围的前缀和。参数 `x` 表示要求解前缀和的位置。函数使用一个循环将 `x` 的所有对应位置上的值累加起来，然后通过 `lowbit` 函数找到 `x` 的上一个位置（即最后一个需要累加的位置），再将该位置上的值累加起来，然后继续找上一个位置，直到累加完所有需要累加的位置为止。这样就得到了从位置 `1` 到位置 `x` 的前缀和。

这就是树状数组的核心操作，通过使用这三个函数，我们可以实现树状数组的各种功能，如求解区间和、单点修改等。

```
1 n, MOD, ans = int(input()), 998244353, 1           # 内存69832KB, 时间2847ms
2 a, fac = list(map(int, input().split())), [1]
3
4 tree = [0] * (n + 1)
5
6 def lowbit(x):
7     return x & -x
8
9 def update(x, y):
10    while x <= n:
11        tree[x] += y
12        x += lowbit(x)
13
14 def getsum(x):
15    tot = 0
16    while x:
17        tot += tree[x]
18        x -= lowbit(x)
19    return tot
20
21
22 for i in range(1, n):
```

```

23     fac.append(fac[i-1] * i % MOD)
24
25 for i in range(1, n + 1):
26     cnt = getsum(a[i-1])
27     update(a[i-1], 1)
28     ans = (ans + ((a[i-1] - 1 - cnt) * fac[n - i]) % MOD) % MOD
29
30 print(ans)

```

状态: Accepted

源代码

```

n, p, ans = int(input()), 998244353, 1
a, fac = list(map(int, input().split())), [1]
c = [0] * (n + 1)

def lowbit(x):
    return x & -x
def getsum(x):
    tot = 0
    while x:
        tot += c[x]
        x -= lowbit(x)
    return tot
def update(x):
    while x <= n:
        c[x] += 1
        x += lowbit(x)

for i in range(1, n):
    fac.append(fac[i - 1] * i % p)
for i in range(n):
    cnt = getsum(a[i])
    update(a[i] + 1)
    ans = (ans + (a[i] - cnt - 1) * fac[n - i - 1]) % p
print(ans)

```

基本信息

#: 42026250
 题目: 27018
 提交人: 张展皓(1779390047)
 内存: 66568kB
 时间: 2846ms
 语言: Python3
 提交时间: 2023-10-27 23:40:18

线段树 (Segment tree)

线段树 segment tree 来计算第*i*位右边比该数还要小的数的个数。

```

1 n, MOD, ans = int(input()), 998244353, 1           # 内存69900KB, 时间5162ms
2 a, fac = list(map(int, input().split())), [1]
3
4 tree = [0] * (2*n)
5
6
7 def build(arr):
8
9     # insert leaf nodes in tree
10    for i in range(n):
11        tree[n + i] = arr[i]
12
13    # build the tree by calculating parents

```

```

14     for i in range(n - 1, 0, -1):
15         tree[i] = tree[i << 1] + tree[i << 1 | 1]
16
17
18 # function to update a tree node
19 def updateTreeNode(p, value):
20
21     # set value at position p
22     tree[p + n] = value
23     p = p + n
24
25     # move upward and update parents
26     i = p
27     while i > 1:
28
29         tree[i >> 1] = tree[i] + tree[i ^ 1]
30         i >>= 1
31
32
33 # function to get sum on interval [l, r)
34 def query(l, r):
35
36     res = 0
37
38     l += n
39     r += n
40
41     while l < r:
42
43         if (l & 1):
44             res += tree[l]
45             l += 1
46
47         if (r & 1):
48             r -= 1
49             res += tree[r]
50
51         l >>= 1
52         r >>= 1
53
54     return res
55
56
57 #build([0]*n)
58
59 for i in range(1, n):
60     fac.append(fac[i-1] * i % MOD)
61
62 for i in range(1, n + 1):
63     cnt = query(0, a[i-1])
64     updateTreeNode(a[i-1]-1, 1)
65

```

```
66     ans = (ans + (a[i-1] -1 - cnt) * fac[n - i]) % MOD
67
68 print(ans)
69
```

状态: Accepted

源代码

```
n, MOD, ans = int(input()), 998244353, 1
a, fac = list(map(int, input().split())), [1]

tree = [0] * (2*n)

def build(arr):

    # insert leaf nodes in tree
    for i in range(n):
        tree[n + i] = arr[i]

    # build the tree by calculating parents
    for i in range(n - 1, 0, -1):
        tree[i] = tree[i << 1] + tree[i << 1 | 1]

# function to update a tree node
def updateTreeNode(p, value):

    # set value at position p
    tree[p + n] = value
    p = p + n

    # move upward and update parents
    i = p
    while i > 1:
        tree[i >> 1] = tree[i] + tree[i ^ 1]
        i >>= 1

# function to get sum on interval [l, r)
def query(l, r):

    res = 0

    l += n
    r += n

    while l < r:

        if (l & 1):
            res += tree[l]
            l += 1

        if (r & 1):
            r -= 1
            res += tree[r]

        l >>= 1
        r >>= 1

    return res

#build([0]*n)

for i in range(1, n):
    fac.append(fac[i-1] * i % MOD)

for i in range(1, n + 1):
    cnt = query(0, a[i-1])
    updateTreeNode(a[i-1]-1, 1)

    ans = (ans + (a[i-1] - 1 - cnt) * fac[n - i]) % MOD

print(ans)
```

基本信息

#: 42078650
题目: 27018
提交人: HFYan(GMyhf)
内存: 69900kB
时间: 5162ms
语言: Python3
提交时间: 2023-10-29 16:12:57

附录B

组合数学是对于计数问题的研究，数论就是对于整除性问题的研究，组合与数论是程序中的常见考点。题目背景知识，数学思维。

因为整数除法具有分配律的性质，单项整除可以等价于各项求和最后整除。

B1 读题

545C. Woodcutters

dp/greedy, 1500, <https://codeforces.com/problemset/problem/545/C>

Little Susie listens to fairy tales before bed every day. Today's fairy tale was about wood cutters and the little girl immediately started imagining the choppers cutting wood. She imagined the situation that is described below.

There are n trees located along the road at points with coordinates x_1, x_2, \dots, x_n . Each tree has its height h_i . Woodcutters can cut down a tree and fell it to the left or to the right. After that it occupies one of the segments $[x_i - h_i, x_i]$ or $[x_i, x_i + h_i]$. The tree that is not cut down occupies a single point with coordinate x_i . Woodcutters can fell a tree if the segment to be occupied by the fallen tree doesn't contain any occupied point. The woodcutters want to process as many trees as possible, so Susie wonders, what is the maximum number of trees to fell.

Input

The first line contains integer n ($1 \leq n \leq 10^5$) — the number of trees.

Next n lines contain pairs of integers x_i, h_i ($1 \leq x_i, h_i \leq 10^9$) — the coordinate and the height of the i -th tree.

The pairs are given in the order of ascending x_i . No two trees are located at the point with the same coordinate.

Output

Print a single number — the maximum number of trees that you can cut down by the given rules.

Examples

input

1	5
2	1 2
3	2 1
4	5 10
5	10 9
6	19 1

output

input

1	5
2	1 2
3	2 1
4	5 10
5	10 9
6	20 1

output

1	4
---	---

Note

In the first sample you can fell the trees like that:

- fell the 1-st tree to the left — now it occupies segment [-1;1]
- fell the 2-nd tree to the right — now it occupies segment [2;3]
- leave the 3-rd tree — it occupies point 5
- leave the 4-th tree — it occupies point 10
- fell the 5-th tree to the right — now it occupies segment [19;20]

In the second sample you can also fell 4-th tree to the right, after that it will occupy segment [10;19].

2023fall-cs101: Algo DS (325)



数院胡睿诚

确实，这个是关键

闫先生: The tree not down occupy a point

1793C. Dora and Search

constructive algorithms, data structures, two pointers, 1200,

<https://codeforces.com/problemset/problem/1793/C>

As you know, the girl Dora is always looking for something. This time she was given a permutation, and she wants to find such a subsegment of it that none of the elements at its ends is either the minimum or the maximum of the entire subsegment. More formally, you are asked to find the numbers l and r ($1 \leq l \leq r \leq n$) such that $a_l \neq \min(a_l, a_{l+1}, \dots, a_r)$, $a_l \neq \max(a_l, a_{l+1}, \dots, a_r)$ and $a_r \neq \min(a_l, a_{l+1}, \dots, a_r)$,

$ar \neq \max(a_l, a_{l+1}, \dots, a_r)$.

A permutation of length n is an array consisting of n distinct integers from 1 to n in any order. For example, [2,3,1,5,4] is a permutation, but [1,2,2] is not a permutation (2 occurs twice in the array) and [1,3,4][1,3,4] is also not a permutation ($n=3$, but 4 is present in the array).

Help Dora find such a subsegment, or tell her that such a subsegment does not exist.

Input

Each test consists of multiple test cases. The first line contains a single integer t ($1 \leq t \leq 10^4$) — the number of test cases. Description of the test cases follows.

For each test case, the first line contains one integer n ($1 \leq n \leq 2 \cdot 10^5$) — the length of permutation.

The second line contains n distinct integers a_1, a_2, \dots, a_n ($1 \leq a_i \leq n$) — the elements of permutation.

It is guaranteed that the sum of n over all test cases doesn't exceed $2 \cdot 1052 \cdot 105$.

Output

For each test case, output -1 if the desired subsegment does not exist.

Otherwise, output two indexes l, r such that $[a_l, a_{l+1}, \dots, a_r]$ satisfies all conditions.

If there are several solutions, then output any of them.

Example

input

```
1 4
2 3
3 1 2 3
4 4
5 2 1 4 3
6 7
7 1 3 2 4 6 5 7
8 6
9 2 3 6 5 4 1
```

output

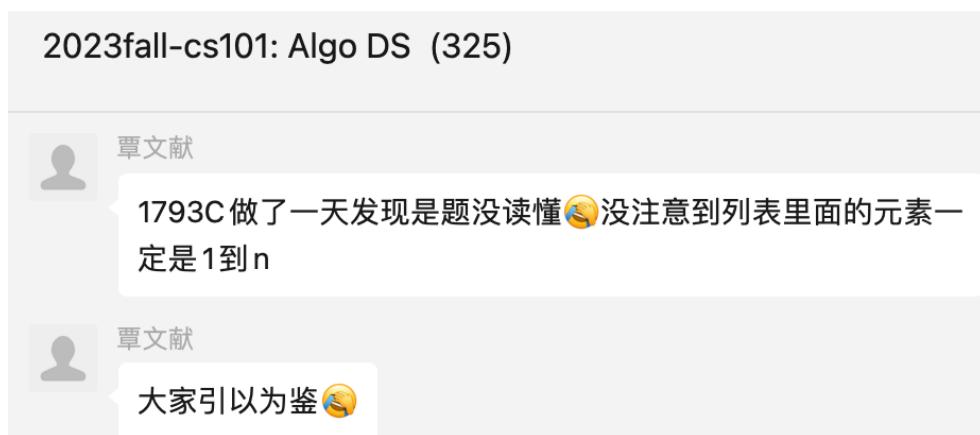
```
1 -1
2 1 4
3 2 6
4 -1
```

Note

In the first and fourth test cases, it can be shown that there are no desired subsegments.

In the second test case, the subsegment [1,4] satisfies all the conditions, because $\max(a_1, a_2, a_3, a_4) = 4$, $\min(a_1, a_2, a_3, a_4) = 1$, as we see, all the conditions are met.

In the third test case, the subsegment [2,6] also satisfies all the conditions described.



数学思维



夏天明

本质上是 dedekind 分割

Yesterday 09:46



数院胡睿诚

理论上肯定是应该从中间扫的 😂😂😂



数院胡睿诚

算是吧，其实不太是

23 元培 夏天明: 本质上是 dedekind 分割



蒋子轩 23 工学院

Accepted	217 ms
Accepted	155 ms



蒋子轩 23 工学院

从中间扫设计的好的话可以比标答快一点



蒋子轩 23 工学院

但差别不大

Yesterday 09:51



夏天明

应该说数据设计的好的话可以让中间扫快一点



数院胡睿诚

其实差不多



数院胡睿诚

因为扫的量级比起排序量级都是忽略的

戴德金原理 (Dedekind principle) 亦称[戴德金分割](#), 是保证直线连续性的基础, 其内容为: 如果把直线的所有点分成两类, 使得: 1.每个点恰属于一个类, 每个类都不空。2.第一类的每个点都在第二类的每个点的前面, 那么, 或者在第一类里存在着这样的点, 第一类中所有其余的点都在它的前面; 或者在第二类里存在着这样的点, 它在第二类的所有其余的点的前面 [3]。这个点决定直线的戴德金割切, 此点称为戴德金点(或界点), 戴德金原理是戴德金 (J.W.)R.Dedekind于1872年提出来的, 在构造欧氏几何的公理系统时, 可以选取它作为连续公理, 在希尔伯特公理组I, II, III的基础上, [阿基米德公理](#)和康托尔公理合在一起与戴德金原理等价。

2023fall-cs101: Algo DS (325) 2023/10/25



数院胡睿诚

先证高度相同不连, 然后分成两类, 一类只连大的, 一类只连小的, 显然两类内部不连 😊

赵时阳-数院 23: 这个应该主要就是从 Turan 的极值出发然后讨论去证吧



数院胡睿诚

几乎就完了



赵时阳-数院 23

是的



赵时阳-数院 23

这个考虑主要应该就是从 Turan 定理极值的例子考虑就很顺 😊



数院胡睿诚

反正我调整调了半天 😊😊😊



数院胡睿诚

也能调就是了



数院胡睿诚

确实是这个味

赵时阳-数院 23: 这个考虑主要应该就是从 Turan 定理极值的例子考虑就很顺 😊



赵时阳-数院 23

调整大法好 😊

https://zhuanlan.zhihu.com/p/528662514?utm_id=0

图兰定理 (Turan's graph theorem) 图论 (graph theory) 的一条基本定理，它是极值图论 (extremal graph theory) 的开端。此定理有很多种证明方法，我们将要介绍其中的五个。在陈述图兰定理前，我们先介绍一些背景知识。令 G 为一个简单图 (simple graph)，即不包含多重边 (multiple edges) 也不包含自环 (loop)。令 G 的顶点 (vertex) 集为 $V = v_1, \dots, v_n$ ，边 (edge) 集为 E 。若一个图的每对顶点都被唯一的一条边相连，则成此图为一个完全图 (complete graph)，而一个图的完全子图 (complete subgraph) 叫作团 (clique)。我们将包含 p 的顶点的 p -团写作 K_p 。

803A. Maximal Binary Matrix

constructive algorithms, 1400, <https://codeforces.com/problemset/problem/803/A>

You are given matrix with n rows and n columns filled with zeroes. You should put k ones in it in such a way that the resulting matrix is symmetrical with respect to the main diagonal (the diagonal that goes from the top left to the bottom right corner) and is lexicographically maximal.

One matrix is lexicographically greater than the other if the first different number in the first different row from the top in the first matrix is greater than the corresponding number in the second one.

If there exists no such matrix then output -1.

Input

The first line consists of two numbers n and k ($1 \leq n \leq 100$, $0 \leq k \leq 106$).

Output

If the answer exists then output resulting matrix. Otherwise output -1.

Examples

input

1	2	1
---	---	---

output

1	1	0
2	0	0

input

1	3	2
---	---	---

output

1	1	0	0
2	0	1	0
3	0	0	0

input

1	2	5
---	---	---

output

1	-1
---	----

2023fall-cs101: Algo DS (325)

...

为什么不可以先把对角线填完? 

张展皓: 但是不应该先把对角线填完



姚哲恒 2300012144

因为要求输出的是最大的矩阵 (



姚哲恒 2300012144

要尽量让靠前的行填满



姚哲恒 2300012144

同时保证对称

"林焯" 拍了拍 "姚哲恒 2300012144"，说六啊



23 何秉儒 物院

比如

1110

1100

1000

0000

比

1100

1100

0010

0001

大

闫先生: 为什么不可以先把对角线填完?

好的。lexicographically maximal. 

B2 题目都有背景知识

12560: 生存游戏

matrices, <http://cs101.openjudge.cn/practice/12560/>

有如下生存游戏的规则：

给定一个 $n*m(1 \leq n, m \leq 100)$ 的数组，每个元素代表一个细胞，其初始状态为活着(1)或死去(0)。

每个细胞会与其相邻的8个邻居（除数组边缘的细胞）进行交互，并遵守如下规则：

任何一个活着的细胞如果只有小于2个活着的邻居，那它就会由于人口稀少死去。

任何一个活着的细胞如果有2个或者3个活着的邻居，就可以继续活下去。

任何一个活着的细胞如果有超过3个活着的邻居，那它就会由于人口拥挤而死去。

任何一个死去的细胞如果有恰好3个活着的邻居，那它就会由于繁殖而重新变成活着的状态。

请写一个函数用来计算所给定初始状态的细胞经过一次更新后的状态是什么。

注意：所有细胞的状态必须同时更新，不能使用更新后的状态作为其他细胞的邻居状态来进行计算。

输入

第一行为 n 和 m ，而后 n 行，每行 m 个元素，用空格隔开。

输出

n 行，每行 m 个元素，用空格隔开。

样例输入

1	3 4
2	0 0 1 1
3	1 1 0 0
4	1 1 0 1

样例输出

1	0 1 1 0
2	1 0 0 1
3	1 1 1 0

来源：cs10116 final exam

康威生命游戏(Game of Life) <https://baike.baidu.com/item/> 康威生命游戏/22668799?fr=ge_ala

康威生命游戏(Game of Life), 剑桥大学约翰·何顿·康威设计的计算机程序。 [1]

美国趣味数学大师马丁·加德纳(Martin Gardner, 1914-2010) 通过《科学美国人》杂志, 将康威的生命游戏介绍给学术界之外的广大读者, 一时吸引了各行各业一大批人的兴趣, 这时细胞自动机课题才吸引了科学家的注意。 [1]

中文名	康威生命游戏	外文名	Conway life game
提出者	约翰·何顿·康威		

目录	1 概述
	2 生存定律

概述

 播报  编辑

生命游戏没有游戏玩家各方之间的竞争, 也谈不上输赢, 可以把它归类为仿真游戏。事实上, 也是因为它模拟和显示的图像看起来颇似生命的出生和繁衍过程而得名为“生命游戏”。在游戏进行中, 杂乱无序的细胞会逐渐演化出各种精致、有形的结构; 这些结构往往有很好的对称性, 而且每一代都在变化形状。一些形状一经锁定就不会逐代变化。有时, 一些已经成形的结构会因为一些无序细胞的“入侵”而被破坏。但是形状和秩序经常能从杂乱中产生出来。

每个方格中都可放置一个生命细胞, 每个生命细胞只有两种状态:

“生”或“死”。用黑色方格表示该细胞为“生”, 空格(白色)表示该细胞为“死”。或者说方格网中黑色部分表示某个时候某种“生命”的分布图。生命游戏想要模拟的是: 随着时间的流逝, 这个分布图将如何一代一代地变化。 [1]

生存定律

 播报  编辑

游戏开始时, 每个细胞随机地设定为“生”或“死”之一的某个状态。然后, 根据某种规则, 计算出下一代每个细胞的状态, 画出下一代细胞的生死分布图。

应该规定什么样的迭代规则呢?需要一个简单的, 但又反映生命之间既协同又竞争的生存定律。为简单起见, 最基本的考虑是假设每一个细胞都遵循完全一样的生存定律; 再进一步, 把细胞之间的相互影响只限制在最靠近该细胞的8个邻居中。

也就是说, 每个细胞迭代后的状态由该细胞及周围8个细胞状态所决定。作了这些限制后, 仍然还有很多方法来规定“生存定律”的具体细节。例如, 在康威的生命游戏中, 规定了如下生存定律。

(1)当前细胞为死亡状态时, 当周围有3个存活细胞时, 则迭代后该细胞变成存活状态(模拟繁殖); 若原先为生, 则保持不变。

(2)当前细胞为存活状态时, 当周围的邻居细胞低于两个(不包含两个)存活时, 该细胞变成死亡状态(模拟生命数量稀少)。

(3)当前细胞为存活状态时, 当周围有两个或3个存活细胞时, 该细胞保持原样。

(4)当前细胞为存活状态时, 当周围有3个以上的存活细胞时, 该细胞变成死亡状态(模拟生命数量过多)。

可以把最初的细胞结构定义为种子, 当所有种子细胞按以上规则处理后, 可以得到第1代细胞图。按规则继续处理当前的细胞图, 可以得到下一代的细胞图, 周而复始。

上面的生存定律当然可以任意改动, 发明出不同的“生命游戏”。 [1]

细胞自动机

[编辑](#)

本词条由“匿名用户”建档。

目录

[1 细胞自动机](#)

[2 概览](#)

细胞自动机

[编辑](#)

元胞自动机 (pl.cellular automata, 缩写CA) 是自动机理论中研究的一种离散计算模型。元胞自动机也称为元胞空间、镶嵌自动机、均质结构、元胞结构、镶嵌结构和迭代阵列。元胞自动机已在各个领域得到应用，包括物理学、理论生物学和微观结构建模。

元胞自动机由规则的细胞网格组成，每个细胞处于有限数量的状态之一，例如开和关（与耦合映射格相反）。网格可以是任意有限维数。对于每个像元，一组称为其邻域的像元是相对于指定像元定义的。通过为每个单元格分配一个状态来选择初始状态（时间 $t = 0$ ）。根据一些固定规则（通常是数学函数）创建新的一代（将 t 推进 1），该规则根据单元格的当前状态及其邻域单元格的状态确定每个单元格的新状态。通常，更新单元格状态的规则对于每个单元格都是相同的，并且不会随时间改变，并且会同时应用于整个网格，但也有例外，例如随机元胞自动机和异步元胞自动机。

这个概念最初是在 1940 年代由 Stanislaw Ulam 和 John von Neumann 在洛斯阿拉莫斯国家实验室同时发现的。虽然在整个 1950 年代和 60 年代都有一些人进行研究，但直到 1970 年代和康威的二维元胞自动机“生命游戏”，人们对该主题的兴趣才扩展到学术界之外。20 世纪 80 年代，Stephen Wolfram 从事一维元胞自动机的系统研究，他称之为初等元胞自动机；他的研究助理 Matthew Cook 证明其中一条规则是图灵完备的。

正如 Wolfram 概述的那样，元胞自动机的主要分类编号为 1 到 4。按顺序，它们是模式通常稳定为同质性的自动机，模式演化为基本稳定或振荡结构的自动机，模式以看似混乱的方式演化的自动机，以及模式变得极其复杂并可能持续一段时间的自动机。最后一类被认为是计算通用的，或者能够模拟图灵机。特殊类型的元胞自动机是可逆的，其中只有一个配置直接导致后续配置，并且是极权的，其中单个单元的未来值仅取决于一组相邻单元的总值。元胞自动机可以模拟各种真实世界的系统，包括生物和化学系统。

基本介绍

[▷ 播报](#)
[编辑](#)

不同于一般的[动力学模型](#)，元胞自动机不是由严格定义的物理方程或函数确定，而是用一系列模型构造的规则构成。凡是满足这些规则的模型都可以算作是元胞自动机模型。因此，元胞自动机是一类模型的总称，或者说是一个方法框架。其特点是时间、空间、状态都离散，每个变量只取有限多个状态，且其状态改变的规则在时间和空间上都是局部的。

通俗解释

[▷ 播报](#)
[编辑](#)

元胞自动机是一个像图1中所示的那种灯泡阵列，每个灯有“开”和“关”两种状态，每个灯与周围的8个灯相连（边上的灯会认为与另一边的相连，比如最左边的灯，会认为与最右边的灯相连。这样所有灯都与8个灯相连）。初始阶段，部分灯开部分灯关。元胞自动机像CPU一样一步一步地进行计算（如果不理解，可以参考[刘慈欣《三体》里的人列计算机](#)，有人喊号子让大家一步一步地动作）。每个元胞自动机有一个规则，来说明每个灯怎么根据之前周围8个灯及自己的状态决定自己下一步时的状态（比如一种规则可以是：采用邻域占多数的状态。图1中即展示了这种规则下下一步此元胞自动机会怎么变化）。这种计算模型，是冯诺依曼提出的，称为冯诺依曼结构。与[图灵机](#)的计算能力是等价的。

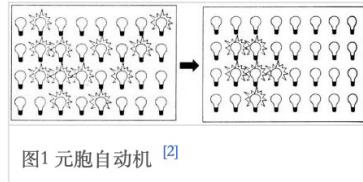


图1 元胞自动机 [2]

(思想与图的来源：[梅拉妮米歇尔《复杂》2008](#) [2]。这也许只是元胞自动机的一种)

具体解释

[▷ 播报](#)
[编辑](#)

元胞自动机的构建没有固定的[数学公式](#)，构成方式繁杂，变种很多，行为复杂。故其分类难度也较大，自元胞自动机产生以来，对于元胞自动机分类的研究就是元胞自动机的一个重要的研究课题和核心理论，在基于不同的出发点，元胞自动机可有多种分类，其中，最具影响力的当属S. Wolfram在80年代初做的基于动力学行为的元胞自动机分类，而基于维数的元胞自动机分类也是最简单和最常用的划分。除此之外，在1990年，Howard A. Gutowicz提出了基于元胞自动机行为的马尔科夫概率量测的层次化、[参量化的](#)分类体系 (Gutowicz, H. A., 1990)。下面就上述的前两种分类作进一步的介绍。同时就几种特殊类型的元胞自动机进行介绍和探讨S. Wolfram在详细[分析研究](#)了一维元胞自动机的演化行为，并在大量的[计算机实验](#)的基础上，将所有元胞自动机的动力学行为归纳为四大类 (Wolfram. S., 1986):

(1) 平稳型：自任何初始状态开始，经过一定时间运行后，元胞空间趋于一个空间平稳的[构形](#)，这里空间平稳即指每一个元胞处于固定状态。不随时间变化而变化。

(2) 周期型：经过一定时间运行后，元胞空间趋于一系列简单的固定结构 (Stable Patterns) 或周期结构 (Periodical Patterns)。由于这些结构可看作是一种[滤波器](#) (Filter)，故可应用到图像处理的研究中。

(3) 混沌型：自任何初始状态开始，经过一定时间运行后，元胞自动机表现出混沌的非周期行为，所生成的结构的统计特征不再变止，通常表现为分形分维特征。

(4) 复杂型：出现复杂的局部结构，或者说是局部的混沌，其中有些会不断地传播。

B3 语法

B3.1 逻辑删除

在Python中，执行删除操作通常建议使用logic删除，而不是physic删除。也就是说，不直接从列表中删除元素，而是标记它已经删除了。删除操作消耗的时间更少。

布尔代数 (Boolean Algebra) 是一种数学上的代数系统，用于处理逻辑运算和关系。它涉及布尔值 (true 和 false) 以及与、或、非等逻辑运算符。在逻辑删除中，布尔代数的概念被用于标记和操作数据行的删除状态，因此可以说逻辑删除涉及布尔代数的使用。

布尔代数

世界上不可能有比二进制更简单的计数方法了，它只有两个数字:0和1。从单纯数学的角度讲，它甚至比我们的十进制更合理。但是我们人有十个手指，使用起来比二进制(或者八进制)方便得多，所以在进化和文明发展过程中人类采用了十进制。二进制的历史其实也很早，中国古代的阴阳学说可以认为是最早二进制的雏形。而二进制作为一个计数系统公元前 2-5 世纪时由印度学者完成，但是他们没有使用0和 1计数。到17 世纪，德国伟大的数学家莱布尼兹(Gottfried Leibniz)把它完善，并且用0和 1表示它的两个数字，成为我们今天使用的二进制。二进制除了是一种计数的方式外，它还可以表示逻辑的“是”与“非”。这第二个特性在索引中非常有用。布尔运算是针对二进制，尤其是二进制第二个特性的运算，它很简单，可能没有比布尔运算更简单的运算。尽管今天每个搜索引擎都声称自己如何聪明、多么智能(这个词非常忽悠人)其实从根本上讲都没有逃出布尔运算的框框。

布尔(George Boole)是19 世纪英国的一位中学数学老师，还创办过一所中学。后来在爱尔兰科克(Cork)的一所学院当教授。生前没有人认为他是数学家，虽然他曾经在剑桥大学数学杂志(Cambridge Mathematical Journal)上发表过论文。(英国另一位生前没有被公认为科学家的是著名物理学家焦耳，虽然他生前已经是英国皇家科学院院士，但是他的公认身份是啤酒商。)布尔在工作之余，喜欢阅读数学论著，思考数学问题。1854 年，布尔的《思维规律》(An Investigation of the Laws of Thought, on which are founded the Mathematical Theories of Logic and Probabilities)一书，第一次向人们展示了如何用数学的方法解决逻辑问题。在此之前，人们普遍的认识是数学和逻辑是两个不同的学科今天联合国教科文组织依然把它们严格分开。

布尔代数简单得不能再简单了。运算的元素只有两个:1(TRUE, 真)和0(FALSE, 假)。基本的运算只有“与”(AND)、“或”(OR)和“非”(NOT) 三种(后来发现，这三种运算都可以转换成“与非”AND-NOT 一种运算)。全部运算只用下列几张真值表就能完全描述清楚。

表 8.1 与运算真值表

AND	1	0
1	1	0
0	0	0

表 8.1 说明，如果 AND 运算的两个元素有一个是 0，则运算结果总是 0。如果两个元素都是 1，运算结果是 1。例如，“太阳从西边升起”这个判断是假的(0)，“水可以流动”这个判断是真的(1)，那么，“太阳从西边升起并且水可以流动”就是假的(0)。

表 8.2 或运算真值表

OR	1	0
1	1	1
0	1	0

表 8.2 说明，如果 OR 运算的两个元素有一个是 1，则运算结果总是 1。

如果两个元素都是 0，则运算结果是 0。比如说，“张三是比赛第一名”“李四是比赛第一名”是真的(1)，那么“张这个结论是假的(0) 三或者李四是第一名”就是真的(1)。

表8.3非运算真值表

NOT	
1	0
0	1

表 8.3 说明，NOT 运算把 1 变成 0，把 0 变成 1。比如，如果“象牙是白的”是真的(1)，那么“象牙不是白的”必定是假的(0)。

这么简单的理论能解决什么实际问题。和布尔同时代的数学家们也有同样的疑问。事实上，在布尔代数提出后 80 多年里，它确实没有什么像样的应用，直到 1938 年香农在他的硕士论文中指出用布尔代数来实现开关电路，才使得布尔代数成为数字电路的基础。所有的数学和逻辑运算，加、减、乘、除、乘方、开方等等，全都能转换成二值的布尔运算。数学的发展实际上是不断地抽象和概括的过程，这些抽象了的方法看似离生活越来越远，但是它们最终能找到适用的地方，布尔代数便是如此。

if 控制语句在程序中用于根据条件的真假来进行逻辑推理和计算，并根据条件的结果选择性地执行特定的代码块。, 就是逻辑推理与计算合二为一。

B3.2 高效数组，array类

<https://bajjiahao.baidu.com/s?id=1770291275843443574&wfr=spider&for=pc>

```
1 import sys
2 import array
3
4 a = array.array('i', [0]*1000000)
5 size = sys.getsizeof(a)//(1024*1024) + 1
6
7 print(f'signed int: {size}MB')
8
9 b = [0]*1000000
10 size = sys.getsizeof(b)//(1024*1024) + 1
11 print(f'list: {size}MB')
12
13 for code in array.typecodes:
14     arr = array.array(code)
15     print(code, arr.itemsize)
```

```
1 signed int: 4MB
2 list: 8MB
3 b 1
4 B 1
5 u 4
```

```
6 h 2
7 H 2
8 i 4
9 I 4
10 l 8
11 L 8
12 q 8
13 Q 8
14 f 4
15 d 8
```

B3.3 OrderedDict

在Python的标准库`collections`中，`OrderedDict`是一种特殊的字典类型，它保留了键值对的插入顺序。

`OrderedDict`在很多情况下提供了与普通字典相似的功能，但在某些特定操作上有所不同，特别是关于顺序的操作。

关于`OrderedDict`删除元素的复杂度，官方文档指出，删除元素的复杂度确实是O(1)。具体来说，删除一个键值对的时间复杂度是O(1)，无论该键值对位于字典的哪个位置。

以下是一个示例，展示了如何使用`OrderedDict`并删除其中的元素：

```
1 from collections import OrderedDict
2
3 # 创建一个OrderedDict
4 od = OrderedDict()
5 od['a'] = 1
6 od['b'] = 2
7 od['c'] = 3
8
9 print("初始OrderedDict:", od)
10
11 # 删除一个元素
12 del od['b']
13 print("删除'b'后的OrderedDict:", od)
14
15 # 删除最后一个元素
16 od.popitem()
17 print("删除最后一个元素后的OrderedDict:", od)
18
19 # 删除第一个元素
20 od.popitem(last=False)
21 print("删除第一个元素后的OrderedDict:", od)
```

关于删除操作的复杂度：

- `del od[key]`：删除指定键的元素，时间复杂度为O(1)。

- `od.pop(key)`：删除并返回指定键的元素，时间复杂度为O(1)。
- `od.popitem(last=True)`：删除并返回最后一个键值对（默认行为），时间复杂度为O(1)。
- `od.popitem(last=False)`：删除并返回第一个键值对，时间复杂度为O(1)。

示例解释：

1. 创建 `OrderedDict`：首先创建一个 `OrderedDict` 并插入一些键值对。
2. 删除指定键的元素：使用 `del` 关键字删除键为 'b' 的元素。
3. 删除最后一个元素：使用 `popitem()` 方法删除并返回最后一个键值对。
4. 删除第一个元素：使用 `popitem(last=False)` 方法删除并返回第一个键值对。

通过这些操作，你可以看到 `OrderedDict` 在删除元素时的高效性。这使得 `OrderedDict` 在需要保持插入顺序并且频繁进行插入和删除操作的场景中非常有用。